SAI VIDYA INSTITUTE OF TECHNOLOGY

(Affiliated to VTU, Belagavi, Approved by AICTE, New Delhi and Govt. of Karnataka)
Accredited by NBA, New Delhi (CSE, ISE, ECE, MECH, CIVIL), NAAC-'A' Grade
Rajanukunte, Bengaluru – 560 064
Tel: 080-2846 8196, email: hodaiml@saividya.ac.in web: www.saividya.ac.in

MOTTO

"Learn to lead"

VISION

Contribute dedicated, skilled, intelligent engineers and business administrators to architect strong India and the world.

MISSION

To impart quality technical education and higher moral ethics associated with skilled training to suit the modern-day technology with innovative concepts, so as to learn to lead the future with full confidence.

OPERATING SYSTEMS LABORATORY

(BCS303)

(As per Visvesvaraya Technological University Syllabus)

Compiled by:

Miss Jayshri
Assistant Professor
Dept. of CSE (AI & ML)

Mrs Shilpa Patil
Assistant Professor
Dept. of CSE (AI & ML)

Name:_____

USN :_____

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
(Artificial Intelligence & Machine Learning)

2023-24

Disclaimer

Trademark

Edition: 2023-24

Document Owners

The primary contacts for questions regarding this document are:

Authors: 1. Prof. Jayshri
2. Prof. Shilpa Patil

Department: Computer Science & Engineering (AI & ML)

Contact email id: **1. jayshri.j@saividya.ac.in**
2. shilpa.p@saividya.ac.in

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING (AI & ML)

VISION

To develop competent and socially responsible Engineers in the domain of Artificial Intelligence and Machine Learning to architect Strong India and the world.

MISSION

To provide Progressive Education and skills in the area of Artificial Intelligence and Machine Learning.

To encourage research in Frontier areas of Artificial Intelligence and to foster an environment that supports professional ethics.

To provide the competency through Industry collaboration in order to prepare the students to face real-world challenges and to develop Entrepreneur skills.

PROGRAM EDUCATIONAL OBJECTIVE

PEO 1: *Analyze, Develop and Apply Innovative ideas to solve real-world problems using Artificial Intelligence and Machine Learning techniques.*

PEO 2: *Pursue higher studies, Graduates will have the ability to contribute novel and scientific research-oriented methods in the Artificial Intelligence and Machine Learning area.*

PEO 3: *Excellence in Entrepreneurial Skills, professionalism, moral and ethical conduct, understanding of social context, and strong communication skills to meet the industry expectancy.*

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, 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 and receive 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 context of technological change.

Program Specific Outcomes

PSO 1: Apply the skills of core computer science engineering, artificial intelligence, machine learning, deep learning to solve futuristic problems.

PSO 2: Demonstrate computer knowledge, practical competency and innovative ideas in computer science engineering, artificial intelligence and machine learning using machine tools and techniques.

OPERATING SYSTEM LABORATORY

Subject code:BCS303 **IA Marks:25**
Hour/week:02 Total Hours:20

Course Outcomes:

At the end of the course, the student will be able to:

CO1 – **Identify** the functionalities of OS and their categories.
CO2 – **Evaluate** multithread techniques and process scheduling algorithms.
CO3 – **Demonstrate** suitable techniques for resource management
CO4 – **Evaluate** file system allocation and memory management techniques architecture.
CO5 – **Review** the protection mechanisms in processing environment.

Sl No.	List of Experiment	CO's	Pageno
1	**Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)**	CO1	3-7
2	**Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.**	CO2	8-16
3	**Develop a C program to simulate producer-consumer problem using semaphores.**	CO3	17-19
4	**Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.**	CO1	20-21
5	**Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.**	CO3	22-25
6	**Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.**	CO4	26-31
7	**Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU**	CO4	32-36
8	**Simulate following File Organization Techniques a) Single level directory b) Two level directory**	CO5	37-44
9	**Develop a C program to simulate the Linked file allocation strategies.**	CO5	45-46
10	**Develop a C program to simulate SCAN disk scheduling algorithm**	CO2	47-49

CONTENT BEYOND SYLLABUS

SIMPLE SHELL PROGRAMS:

Sl No.	List of Experiment	CO's	Page no
1	**Write a Shell program to check the given number is even or odd**	CO1	50
2	**Write a Shell program to check the given year is leap year or not**	CO1	51
3	**Write a Shell program to find the factorial of a number**	CO1	52
4	**Write a Shell program to swap the two integers**	CO1	53

1. Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)

i) fork()

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>

 int main(int argc, char **argv)
 {
        pid_t pid;
        pid = fork();
        if(pid==0)
        {
                printf("It is the child process and pid is %d\n",getpid());
                exit(0);
         }

        else if(pid > 0)
        {
                printf("It is the parent process and pid is %d\n",getpid());
        }

        else
        {
                printf("Error while forking\n");
                exit(EXIT_FAILURE);
        }
 return 0;
  }
```

Output:

$make fork

cc fork.c -o fork

And running the script, we get the result as below screenshot.

$ ./fork

It is the parent process and pid is 18097

It is the child process and pid is 18098

ii) exec()

In C programming on Linux and Ubuntu, consider the following example: We have two c files example.c and hello.c.

Code

example.c

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

int main( int argc, char *argv[] )
{
        printf("PID of example.c = %d\n", getpid());
        char *args[] = {"Hello", "C", "Programming", NULL};
        execv("./hello", args);
        printf("Back to example.c");
        return 0;
}
```

hello.c

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

int main( int argc, char *argv[] )
{
   printf("We are in Hello.c\n");
   printf("PID of hello.c = %d\n", getpid());
   return 0;
}
```

OUTPUT:

PID of example.c = 4733

We are in Hello.c

PID of hello.c = 4733

iii) wait()

```
#include<stdio.h> // printf()
#include<stdlib.h> // exit()
#include<sys/types.h> // pid_t
#include<sys/wait.h> // wait()
#include<unistd.h> // fork

int main(int argc, char **argv)
{

        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\n");
         }

        else
        {
                printf("Error in forking..\n");
                exit(EXIT_FAILURE);
        }

return 0;

}
```

OUTPUT:

$ make wait
cc wait.c -o wait
And running the script, we get the result as below screenshot.

$ ./wait

It is the parent process and pid is 18308

It is the child process and pid is 18309
0
1
2
3
4
5
6
7
child is reaped

iv) Create Process & Terminate Process:

CODE

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

int main()

{
        pid_t pid, mypid, myppid;
        pid = getpid();
        printf("Before fork: Process id is %d\n", pid);
        pid = fork();

        if (pid < 0)
        {
                perror("fork() failure\n");
                return 1;
        } // Child process

        if (pid == 0)
        {
            printf("This is child process\n");

            mypid = getpid();
```

```
            myppid = getppid();
            printf("Process id is %d and PPID is %d\n", mypid, myppid);
        }

        else
        {
            // Parent process
            sleep(2);
            printf("This is parent process\n");
            mypid = getpid();
            myppid = getppid();
            printf("Process id is %d and PPID is %d\n", mypid, myppid);
            printf("Newly created process id or child pid is %d\n", pid);
    }
    return 0;
```

OUTPUT

```
Before fork: Process id is 166629
This is child process
Process id is 166630 and PPID is 166629
Before fork: Process id is 166629
This is parent process
Process id is 166629 and PPID is 166628
Newly created process id or child pid is 1666
```

2. Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a) FCFS:

ALGORITHM:

1. Enter all the processes and their burst time.
2. Find waiting time, **WT** of all the processes.
3. For the 1st process, **WT = 0**.
4. For all the next processes i, **WT[i] = BT[i-1] + WT[i-1]**.
5. Calculate Turnaround **time = WT + BT** for all the processes.
6. Calculate **average waiting time** = total waiting time/no. of processes.
7. Calculate **average turnaround time** = total turnaround time/no. of processes.

CODE:

```
#include <stdio.h>
int main()

{
    int pid[15];
    int bt[15];
    int n;

   printf("Enter the number of processes: ");
   scanf("%d",&n);
   printf("Enter process id of all the processes: ");

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

   for(int i=0;i<n;i++)
   {
      scanf("%d",&bt[i]);
   }

   int i, wt[n];
   wt[0]=0;

  for(i=1; i<n; i++)                          //for calculating waiting time of each process
  {
      wt[i]= bt[i-1]+ wt[i-1];
   }

   printf("Process ID     Burst Time     Waiting Time     TurnAround Time\n");

   float twt=0.0;
```

```
    float tat= 0.0;

    for(i=0; i<n; i++)
    {
        printf("%d\t\t", pid[i]);
        printf("%d\t\t", bt[i]);
        printf("%d\t\t", wt[i]);
        printf("%d\t\t", bt[i]+wt[i]);     //calculating and printing turnaround time of each process
        printf("\n");

        twt += wt[i];                      //for calculating total waiting time
        tat += (wt[i]+bt[i]);              //for calculating total turnaround time

    }
    float att,awt;
    awt = twt/n;                 //for calculating average waiting time
    att = tat/n;                 //for calculating average turnaround time

    printf("Avg. waiting time= %f\n",awt);
    printf("Avg. turnaround time= %f",att);
}
```

OUTPUT:

```
Enter the number of processes: 3
Enter process id of all the processes: 1 2 3
Enter burst time of all the processes: 5 11 11
Process ID     Burst Time     Waiting Time     TurnAround Time
1                   5               0                5
2                  11               5               16
3                  11              16               27
Avg. waiting time= 7.000000
Avg. turnaround time= 16.000000
```

b) SJF

ALGORITHM:

1. Enter number of processes.
2. Enter the **burst time** of all the processes.
3. Sort all the processes according to their **burst time**.
4. Find waiting time, **WT** of all the processes.
5. For the smallest process, **WT = 0**.
6. For all the next processes **i**, find waiting time by adding burst time of all the previously completed process.
7. Calculate **Turnaround time = WT + BT** for all the processes.
8. Calculate **average waiting time = total waiting time / no. of processes**.
9. Calculate **average turnaround time= total turnaround time / no. of processes**.

CODE:

```
#include<stdio.h>

int main()
{
   int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;
   float avg_wt,avg_tat;

   printf("Enter number of process:");
   scanf("%d",&n);
   printf("\nEnter Burst Time:\n");

   for(i=0;i<n;i++)
   {
      printf("p%d:",i+1);
      scanf("%d",&bt[i]);
      p[i]=i+1;
   }

   for(i=0;i<n;i++)                              //sorting of burst times
   {
         pos=i;
         for(j=i+1;j<n;j++)
         {
             if(bt[j]<bt[pos])
             pos=j;
         }

      temp=bt[i];
      bt[i]=bt[pos];
      bt[pos]=temp;

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

   wt[0]=0;

   for(i=1;i<n;i++)                              //finding the waiting time of all the processes
   {
        wt[i]=0;

        for(j=0;j<i;j++)
       {
         wt[i]+=bt[j];      //individual WT by adding BT of all previous completed processes
         total+=wt[i];
       }
   }
```

```
    avg_wt=(float)total/n;                          //average waiting time
    printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];              //turnaround time of individual processes
        totalT+=tat[i];                  //total turnaround time

        printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
     }

      avg_tat=(float)totalT/n;                  //average turnaround time

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

OUTPUT:

Enter number of process:4

Enter Burst Time:
p1:5
p2:4
p3:12
p4:7

Process	Burst Time	Waiting Time	Turnaround Time
p2	4	0	4
p1	5	4	9
p4	7	9	16
p3	12	16	28

Average Waiting Time=7.250000
Average Turnaround Time=14.250000

c) ROUND ROBIN

ALGORITHM:

Step 1: Organize all processes according to their arrival time in the ready queue. The queue structure of the ready queue is based on the FIFO structure to execute all CPU processes.

Step 2: Now, we push the first process from the ready queue to execute its task for a fixed time, allocated by each process that arrives in the queue.

Step 3: If the process cannot complete their task within defined time interval or slots because it is stopped by

another process that pushes from the ready queue to execute their task due to arrival time of the next process is reached. Therefore, CPU saved the previous state of the process, which helps to resume from the point where it is interrupted. (If the burst time of the process is left, push the process end of the ready queue).

Step 4: Similarly, the scheduler selects another process from the ready queue to execute its tasks. When a process finishes its task within time slots, the process will not go for further execution because the process's burst time is finished.

Step 5: Similarly, we repeat all the steps to execute the process until the work has finished.

CODE:

```
#include<stdio.h>
#include<conio.h>
void main()

{
   int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;

   printf(" Total number of process in the system: ");
   scanf("%d", &NOP);
   y = NOP;                        // Assign the number of process to variable y

  // Use for loop to enter the details of the process like Arrival time and the Burst Time

   for (i=0; i<NOP; i++)

  {
        printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
        printf(" Arrival time is: \t");            // Accept arrival time
        scanf("%d", &at[i]);
        printf(" \nBurst time is: \t");            // Accept the Burst time
        scanf("%d", &bt[i]);

        temp[i] = bt[i];                                  // store the burst time in temp array
   }

   printf("Enter the Time Quantum for the process: \t");      // Accept the Time quantum
   scanf("%d", &quant);

  // Display the process No, burst time, Turn Around Time and the waiting time
   printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

   for (sum=0, i = 0; y!=0; )
  {
        if(temp[i] <= quant && temp[i] > 0)            // define the conditions
```

```
        {
                sum = sum + temp[i];
                temp[i] = 0;
               count=1;
        }
        else if(temp[i] > 0)
        {
                temp[i] = temp[i] - quant;
                sum = sum + quant;
        }

    if(temp[i]==0 && count==1)
    {
       y--;                                                    //decrement the process no.
       printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
       wt = wt+sum-at[i]-bt[i];
       tat = tat+sum-at[i];
       count =0;
    }

    if(i==NOP-1)
    {
       i=0;
     }
    else if(at[i+1]<=sum)
    {
       i++;
    }
    else
    {
       i=0;
    }
 }

  avg_wt = wt * 1.0/NOP;     // represents the average waiting time and Turn Around time
  avg_tat = tat * 1.0/NOP;    //  represents the average Turn Around time

  printf("\n Average Turn Around Time: \t%f", avg_wt);
 printf("\n Average Waiting Time: \t%f", avg_tat);
 getch();

}
```

OUTPUT:

Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]

Arrival time is: 0

Burst time is: 8

Enter the Arrival and Burst time of the Process[2]
Arrival time is: 1
Burst time is: 5

Enter the Arrival and Burst time of the Process[3]
Arrival time is: 2
Burst time is: 10

Enter the Arrival and Burst time of the Process[4]
Arrival time is: 3
Burst time is: 11

Enter the Time Quantum for the process: 6

Process No	Burst Time	TAT	Waiting Time
Process No[2]	5	10	5
Process No[1]	8	25	17
Process No[3]	10	27	17
Process No[4]	11	31	20

Average Turn Around Time: 14.750000
Average Waiting Time: 23.25000

d) PRIORITY SCHEDULEING:

ALGORITHM:

The algorithms prioritize the processes that must be carried out and schedule them accordingly. A higher priority process will receive CPU priority first, and this will continue until all of the processes are finished. The algorithm that places the processes in the ready queue in the order determined by their priority and chooses which ones to go to the job queue for execution requires both the job queue and the ready queue. Due to the process' greater priority, the Priority Scheduling Algorithm is in charge of transferring it from the ready queue to the work queue.

CODE:

```
#include <stdio.h>

void swap(int *a,int *b)

{
   int temp=*a;
   *a=*b;
   *b=temp;
}

int main()
{
   int n;
   printf("Enter Number of Processes: ");
   scanf("%d",&n);

   int burst[n],priority[n],index[n];

   for(int i=0;i<n;i++)
   {
         printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
         scanf("%d %d",&burst[i],&priority[i]);
         index[i]=i+1;
   }

   for(int i=0;i<n;i++)
   {
        int temp=priority[i],
        int m=i;

    for(int j=i;j<n;j++)
    {
        if(priority[j] > temp)
        {
           temp=priority[j];
           m=j;
        }
     }
     swap(&priority[i], &priority[m]);
     swap(&burst[i], &burst[m]);
     swap(&index[i],&index[m]);
   }

   int t=0;
   printf("Order of process Execution is\n");

   for(int i=0;i<n;i++)
   {
     printf("P%d is executed from %d to %d\n",index[i],t,t+burst[i]);
     t+=burst[i];
```

```
    }

    printf("\n");
    printf("Process Id\tBurst Time\tWait Time\n");

    int wait_time=0;
    int total_wait_time = 0;

    for(int i=0;i<n;i++)
    {
        printf("P%d\t\t%d\t\t%d\n",index[i],burst[i],wait_time);
        total_wait_time += wait_time;
        wait_time += burst[i];
    }

    float avg_wait_time = (float) total_wait_time / n;
    printf("Average waiting time is %f\n", avg_wait_time);

    int total_Turn_Around = 0;

    for(int i=0; i < n; i++)
   {
        total_Turn_Around += burst[i];
    }

    float avg_Turn_Around = (float) total_Turn_Around / n;
    printf("Average TurnAround Time is %f",avg_Turn_Around);
    return 0;
}
```

OUTPUT:

```
Enter Number of Processes: 2
Enter Burst Time and Priority Value for Process 1: 5 3
Enter Burst Time and Priority Value for Process 2: 4 2
Order of process Execution is
P1 is executed from 0 to 5
P2 is executed from 5 to 9
Process Id          Burst Time          Wait Time
P1                  5                   0
P2                  4                   5
Average waiting time is 2.500000
Average TurnAround Time is 4.50
```

3. Develop a C program to simulate producer-consumer problem using semaphores

ALGORITHM:

1. Initialize the empty semaphore to the size of the buffer and the full semaphore to 0.
2. The producer acquires the empty semaphore to check if there are any empty slots in the buffer. If there are no empty slots, the producer blocks until a slot becomes available.
3. The producer acquires the mutex to access the buffer, inserts a data item into an empty slot in the buffer, and releases the mutex.
4. The producer releases the full semaphore to indicate that a slot in the buffer is now full.
5. The consumer acquires the full semaphore to check if there are any full slots in the buffer. If there are no full slots, the consumer blocks until a slot becomes available.
6. The consumer acquires the mutex to access the buffer, reads a data item from a full slot in the buffer, and releases the mutex.
7. The consumer releases the empty semaphore to indicate that a slot in the buffer is now empty.

CODE:

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

int mutex = 1;                                  // Initialize a mutex to 1
int full = 0;                                   // Number of full slots as 0
int empty = 10, x = 0;                          // Number of empty slots as size  of buffer

void producer()                 // Function to produce an item and add it to the buffer
{
   --mutex;                                     // Decrease mutex value by 1
   ++full;                                     // Increase the number of full slots by 1
   --empty;                                   // Decrease the number of empty slots by 1
    x++;                                              // Item produced
    printf("\nProducer produces"item %d", x);

   ++mutex;                                 // Increase mutex value by 1
}

void consumer()                 // Function to consume an item and remove it from buffer
{
   --mutex;                                  // Decrease mutex value by 1
   --full;                                  // Decrease the number of full slots by 1
   ++empty;                                 // Increase the number of empty slots by 1
   printf("\nConsumer consumes item %d", x);
   x--;
   ++mutex;                         // Increase mutex value by 1
}
```

```
int main()                          // Driver Code
{
   int n, i;

   printf("\n1. Press 1 for Producer"
         "\n2. Press 2 for Consumer"
         "\n3. Press 3 for Exit");


   for (i = 1; i > 0; i++)
  {
      printf("\nEnter your choice:");
      scanf("%d", &n);

      switch (n) {

      case 1:

         if ((mutex == 1)&& (empty != 0))
          // If mutex is 1 and empty  is non-zero, then it is possible to produce
         {
            producer();
         }
         else                                // Otherwise, print buffer is full
         {
            printf("Buffer is full!");
         }
         break;

      case 2:

         // If mutex is 1 and full is non-zero, then it is possible to consume
         if ((mutex == 1)  && (full != 0))
         {
            consumer();
         }
         else                                 // Otherwise, print Buffer  is empty
         {
            printf("Buffer is empty!");
         }
         break;

      case 3:                                 // Exit Condition
         exit(0);
         break;
      }
   }
}
```

OUTPUT:

1. Produce 2. Consume 3. Exit
Enter your choice: 2
Buffer is Empty
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce 2. Consume 3. Exit
Enter your choice: 2
The consumed value is 100
1. Produce 2. Consume 3. Exit
Enter your choice: 3

2nd one:
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 300

Enter your choice: 2
The consumed value is 100
1. Produce 2. Consume 3. Exit
Enter your choice: 2
The consumed value is 300

Enter your choice: 3

4. Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.

ALGORITHM:

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we 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.

CODE:

/*writer process */

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd;
    char buf[1024];
   char * myfifo = "/tmp/myfifo";

    mkfifo(myfifo, 0666);
    printf("Run Reader process to read the FIFO File\n");
    fd = open(myfifo, O_WRONLY);
    write(fd,"Hi", sizeof("Hi"));                /* write "Hi" to the FIFO */
```

```
    close(fd);
    unlink(myfifo); /* remove the FIFO */
    return 0;
}
```

OUTPUT:

```
Run Reader process to read the FIFO File
```

/* Reader Process */

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define MAX_BUF 1024

int main()
{
    int fd;
   /* A temp FIFO file is not created in reader */
   char *myfifo = "/tmp/myfifo";
   char buf[MAX_BUF];
   /* open, read, and display the message from the FIFO */
   fd = open(myfifo, O_RDONLY);
   read(fd, buf, MAX_BUF);
   printf("Writer: %s\n", buf);
   close(fd);
   return 0;
}
```

OUTPUT:

Writer: Hi

5. Develop a C program to simulate Bankers Algorithm for Dead Lock Avoidance.

ALGORITHM:

1. Active:= Running U Blocked;

for k=1…r

New_ request[k]:= Requested_ resources[requesting_ process, k];

2. **Simulated_ allocation**:= Allocated_ resources;

for k=1…..r //Compute projected allocation state

Simulated_ allocation [requesting _process, k]:= Simulated_ allocation [requesting _process, k] + New_ request[k];

3. **feasible**:= true;

for k=1….r // Check whether projected allocation state is feasible

0 seconds of 0 secondsVolume 0% if

Total_ resources[k]< Simulated_ total_ alloc [k] then feasible:= false;

4. **if feasible**= true

then // Check whether projected allocation state is a safe allocation state

while set Active contains a process P1 such that

For all k, Total _resources[k] – Simulated_ total_ alloc[k]>= Max_ need [l ,k]-Simulated_ allocation[l, k]

Delete Pl from Active;

for k=1…..r

Simulated_ total_ alloc[k]:= Simulated_ total_ alloc[k]- Simulated_ allocation[l, k];

5. I**f set Active is empty**

then // Projected allocation state is a safe allocation state

for k=1….r // Delete the request from pending requests

Requested_ resources [requesting_ process, k]:=0;

for k=1….r // Grant the request

Allocated_ resources [requesting_ process, k]:= Allocated_ resources[requesting_ process, k] + New_ request[k];

Total_ alloc[k]:= Total_ alloc[k] + New_ request[k];

CODE:

```
#include<stdio.h>
int main()

{
   int n,r,i,j,k,p,u=0,s=0,m;
   int block[10],run[10],active[10],newreq[10];
   int max[10][10],resalloc[10][10],resreq[10][10];
   int totalloc[10],totext[10],simalloc[10];

   printf("Enter the no of processes:");
   scanf("%d",&n);
   printf("Enter the no ofresource classes:");
   scanf("%d",&r);
   printf("Enter the total existed resource in each class:");

   for(k=1; k<=r; k++)
  {
     scanf("%d",&totext[k]);
     printf("Enter the allocated resources:");

   for(i=1; i<=n; i++)
  {
      for(k=1; k<=r; k++)
     {
         scanf("%d",&resalloc[i][k]);
         ("Enter the process making the new request:");
         scanf("%d",&p);
         printf("Enter the requested resource:");

   for(k=1; k<=r; k++)
          scanf("%d",&newreq[k]);
          printf("Enter the process which are n blocked or running:");

   for(i=1; i<=n; i++)
   {
      if(i!=p)
      {
         printf("process %d:\n",i+1);
         scanf("%d%d",&block[i],&run[i]);
      }
   }
   block[p]=0;
   run[p]=0;
   for(k=1; k<=r; k++)
   {
      j=0;



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

```
    {
       totalloc[k]=j+resalloc[i][k];
       j=totalloc[k];
    }
    }
for(i=1; i<=n; i++)
{
   if(block[i]==1||run[i]==1)
      active[i]=1;
   else
      active[i]=0;
}
for(k=1; k<=r; k++)
{
   resalloc[p][k]+=newreq[k];
   totalloc[k]+=newreq[k];
}
for(k=1; k<=r; k++)
{
   if(totext[k]-totalloc[k]<0)
   {
      u=1;
      break;
   }
}

if(u==0)
{
   for(k=1; k<=r; k++)
      simalloc[k]=totalloc[k];
   for(s=1; s<=n; s++)
      for(i=1; i<=n; i++)
      {
         if(active[i]==1)
         {
            j=0;
            for(k=1; k<=r; k++)
            {
               if((totext[k]-simalloc[k])<(max[i][k]-resalloc[i][k]))
               {
                  j=1;
                  break;
               }
            }
         }

         if(j==0)
         {
            active[i]=0;

            for(k=1; k<=r; k++)
```

```
                    simalloc[k]=resalloc[i][k];
                }
            }
        m=0;
        for(k=1; k<=r; k++)
            resreq[p][k]=newreq[k];
            printf("Deadlock willn't occur");
    }

    else
    {
        for(k=1; k<=r; k++)
        {
            resalloc[p][k]=newreq[k];
            totalloc[k]=newreq[k];
        }
        printf("Deadlock will occur");
    }
    return 0;
}
```

OUTPUT:

```
Enter the no of processes:4
Enter the no ofresource classes:3
Enter the total existed resource in each class:3 2 2
Enter the allocated resources:1 0 0 5 1 1 2 1 1 0 0 2
Enter the process making the new request:2
Enter the requested resource:1 1 2
Enter the process which are n blocked or running:process 2:
1 2
process 4:
1 0
process 5:
1 0
Deadlock will occur
```

6. Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.

a) WORST-FIT

ALGORITHM:

In this allocation technique, the process traverses the whole memory and always search for the largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search the largest hole

CODE:

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

int main()
{
    int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
    static int bf[max],ff[max];
    clrscr();
    printf("\n\tMemory Management Scheme - Worst Fit");
    printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the blocks:-\n");

    for(i=1;i<=nb;i++)
    {
        printf("Block %d:",i);
        scanf("%d",&b[i]);
    }

    printf("Enter the size of the files :-\n");

    for(i=1;i<=nf;i++)
   {
      printf("File %d:",i);
      scanf("%d",&f[i]);
   }

    for(i=1;i<=nf;i++)
   {
      for(j=1;j<=nb;j++)
     {
        if(bf[j]!=1) //if bf[j] is not allocated
       {
```

```
            temp=b[j]-f[i];
            if(temp>=0)
            if(highest<temp)
            {
               ff[i]=j;
               highest=temp;
             }
           }
        }
        frag[i]=highest;
        bf[ff[i]]=1;
        highest=0;
    }

printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
return 0;
}
```

OUTPUT:

INPUT
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	3	7	6
2	4	1	5	1

B) BEST-FIT

ALGORITHM:

1. The operating system maintains a list of all free memory blocks available in the system.
2. When a process requests memory, the operating system searches the list for the smallest free block of memory that is large enough to accommodate the process.
3. If a suitable block is found, the process is allocated memory from that block.
4. If no suitable block is found, the operating system can either wait until a suitable block becomes available or request additional memory from the system.
5. The best-fit allocation algorithm has the advantage of minimizing external fragmentation, as it searches for the smallest free block of memory that can accommodate a process. However, it can also lead to more internal fragmentation, as processes may not use the entire memory block allocated to them.

CODE:

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
      int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
      static int bf[max],ff[max];
      clrscr();

      printf("\nEnter the number of blocks:");
      scanf("%d",&nb);
      printf("Enter the number of files:");
      scanf("%d",&nf);
      printf("\nEnter the size of the blocks:-\n");

      for(i=1;i<=nb;i++)
      {
          printf("Block %d:",i);
          scanf("%d",&b[i]);
       }
      printf("Enter the size of the files :-\n");

      for(i=1;i<=nf;i++)
      {
        printf("File %d:",i);
        scanf("%d",&f[i]);
      }
```

```
for(i=1;i<=nf;i++)
{
   for(j=1;j<=nb;j++)
   {
     if(bf[j]!=1)
     {
       temp=b[j]-f[i];
       if(temp>=0)
       if(lowest>temp)
        {
           ff[i]=j;
           lowest=temp;
         }
     }
  }
  frag[i]=lowest;
  bf[ff[i]]=1;
  lowest=10000;
  }

  printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");

  for(i=1;i<=nf && ff[i]!=0;i++)
  {
   printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]));
   getch();
  }
```

OUTPUT:
INPUT
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	2	2	1
2	4	1	5	1

C) FIRST-FIT

ALGORITHM:

1 Input memory blocks with size and processes with size.

2 Initialize all memory blocks as free.

3 Start by picking each process and check if it can be assigned to current block.

4 If size-of-process <= size-of-block if yes then assign and check for next process.

5 If not then keep checking the further blocks.

CODE:

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

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

        printf("\n\tMemory Management Scheme - First Fit");
        printf("\nEnter the number of blocks:");
        scanf("%d",&nb);
        printf("Enter the number of files:");
        scanf("%d",&nf);
        printf("\nEnter the size of the blocks:-\n");

        for(i=1;i<=nb;i++)
        {
          printf("Block %d:",i);
          scanf("%d",&b[i]);
        }
        printf("Enter the size of the files :-\n");

        for(i=1;i<=nf;i++)
        {
          printf("File %d:",i);
          scanf("%d",&f[i]);
        }
        for(i=1;i<=nf;i++)
        {
          for(j=1;j<=nb;j++)
          {
            if(bf[j]!=1)
            {
```

```
                temp=b[j]-f[i];

                if(temp>=0)
                {
                   ff[i]=j;
                   break;
        }
      }
   }
    frag[i]=temp;
    bf[ff[i]]=1;
  }

  printf("\n File_no:\t File_size :\t Block_no:\t Block_size:\t Fragement");

  for(i=1;i<=nf;i++)
          printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
          return 0;
}
```

OUTPUT:

INPUT
Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	1	5	4
2	4	3	7	3

7. Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

a. FIFO

ALGORITHM:

1. Start the process

2. Declare the size with respect to page length

3. Check the need of replacement from the page to memory

4. Check the need of replacement from old page to new page in memory

5. Form queue to hold all pages

6. Insert the page require memory into the queue

7. Check for bad replacement and page fault

8. Get the number of processes to be inserted

9. Display the values

10. Stop the process

CODE:

```
#include<stdio.h>
int main()
{
        int i,j,n,a[50],frame[10],no,k,avail,count=0;

        printf("\n ENTER THE NUMBER OF PAGES:\n");
        scanf("%d",&n);
        printf("\n ENTER THE PAGE NUMBER :\n");

        for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
        printf("\n ENTER THE NUMBER OF FRAMES :");
        scanf("%d",&no);

        for(i=0;i<no;i++)
                frame[i]= -1;
                 j=0;
                 printf("\tref string\t page frames\n")

       for(i=1;i<=n;i++)
        {
             printf("%d\t\t",a[i]);
            avail=0;
```

```
        for(k=0;k<no;k++)
            if(frame[k]==a[i])
                avail=1;
             if (avail==0)
                {
                    frame[j]=a[i];
                    j=(j+1)%no;
                    count++;
                    for(k=0;k<no;k++)
                    printf("%d\t",frame[k]);
        }
                        printf("\n");
        }
                printf("Page Fault Is %d",count);
                return 0;
}
```

OUTPUT:

ENTER THE NUMBER OF PAGES: 20
ENTER THE PAGE NUMBER : 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
ENTER THE NUMBER OF FRAMES : 3

ref string	page frames		
7	7	-1	-1
0	7	0	-1
1	7	0	1
2	2	0	1
0			
3	2	3	1
0	2	3	0
4	4	3	0
2	4	2	0
3	4	2	3
0	0	2	3
3			
2			
1	0	1	3
2	0	1	2
0			
1			
7	7	1	2
0	7	0	2
1	7	0	1

Page Fault is 15

b. 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.

CODE:

```
#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("\n The 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
```

8. Simulate following File Organization Techniques a) Single level directory b) Two level directory.

a. SINGLE LEVEL DIRECTORY

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the count, file name, graphical interface.

Step-3: Read the number of files

Step-4: Read the file name

Step-5: Declare the root directory

Step-6: Using the file eclipse function define the files in a single level

Step-7: Display the files

Step-8: Stop the program

CODE:

```
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir;

void main()
{
    int i,ch;
    char f[30];
    clrscr();
    dir.fcnt = 0;
    printf("\nEnter name of directory -- ");
    scanf("%s", dir.dname);
    while(1)
    {
        printf("\n\n 1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5.      Exit\nEnter your choice
-- ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\n Enter the name of the file -- ");
                    scanf("%s",dir.fname[dir.fcnt]);
                    dir.fcnt++;
                    break;

            case 2: printf("\n Enter the name of the file -- ");
                   scanf("%s",f);
```

```
                for(i=0;i<dir.fcnt;i++)
                {
                   if(strcmp(f, dir.fname[i])==0)
                   {
                         printf("File %s is deleted ",f);
                         strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
                         break;
                    }
                  }
          if(i==dir.fcnt)
          printf("File %s not found",f);
          else
          dir.fcnt--;
          break;

case 3: printf("\n Enter the name of the file -- ");
          scanf("%s",f);
          for(i=0;i<dir.fcnt;i++)
          {
             if(strcmp(f, dir.fname[i])==0)
            {
               printf("File %s is found ", f);
               break;
             }
          }
          if(i==dir.fcnt)
          printf("File %s not found",f);
          break;

case 4: if(dir.fcnt==0)
          printf("\n Directory Empty");
          else
          {
             printf("\n The Files are -- ");
             for(i=0;i<dir.fcnt;i++)
               printf("\t%s",dir.fname[i]);
          }

          break;
          default: exit(0);
          }
}

          getch();
          }
```

OUTPUT:

Enter name of directory -- CSE

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- A

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- B

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 4

The Files are -- A B C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 3

Enter the name of the file – ABC

File ABC not found
1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 2

Enter the name of the file – B
File B is deleted

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 5

b. TWO LEVEL DIRECTORY:

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the count, file name, graphical interface.

Step-3: Read the number of files

Step-4: Read the file name

Step-5: Declare the root directory

Step-6: Using the file eclipse function define the files in a two level diretory.

Step-7: Display the files

Step-8: Stop the program

CODE:

```
#include<stdio.h>
struct
{
   char dname[10],fname[10][10];
   int fcnt;
   dir[10];
}

void main()
{
    int i,ch,dcnt,k;
    char f[30], d[30];
    clrscr();
    dcnt=0;
    while(1)
    {
        printf("\n\n 1. Create Directory\t 2. Create File\t 3. Delete File");
        printf("\n 4. Search File \t \t 5. Display \t 6. Exit \t Enter your choice -- ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\n Enter name of directory -- ");
                    scanf("%s", dir[dcnt].dname);
                    dir[dcnt].fcnt=0;
                    dcnt++;
                    printf("Directory created");
                    break;
           case 2: printf("\n Enter name of the directory -- ");
                   scanf("%s",d);
```

```
            for(i=0;i<dcnt;i++)
            if(strcmp(d,dir[i].dname)==0)
            {
                printf("Enter name of the file -- ");
                scanf("%s",dir[i].fname[dir[i].fcnt]);
                dir[i].fcnt++;
                printf("File created");
                break;
            }

    if(i==dcnt)
    printf("Directory %s not found",d);
    break;

   case 3: printf("\nEnter name of the directory -- ");
            scanf("%s",d);
            for(i=0;i<dcnt;i++)
            {
              if(strcmp(d,dir[i].dname)==0)
              {
                  printf("Enter name of the file -- ");
                  scanf("%s",f);
                  for(k=0;k<dir[i].fcnt;k++)
                  {
                     if(strcmp(f, dir[i].fname[k])==0)
                    {
                            printf("File %s is deleted ",f);
                            dir[i].fcnt--;
                            strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
                            goto jmp;
                    }
                  }
printf("File %s not found",f);
goto jmp;
}
}
printf("Directory %s not found",d);
jmp :
break;

case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
  if(strcmp(d,dir[i].dname)==0)
  {
     printf("Enter the name of the file -- ");
     scanf("%s",f);
     for(k=0;k<dir[i].fcnt;k++)
     {
```

```
            if(strcmp(f, dir[i].fname[k])==0)
            {
               printf("File %s is found ",f);
               goto jmp1;
             }
          }
    printf("File %s not found",f);
    goto jmp1;
  }
}

    printf("Directory %s not found",d);
    jmp1: break;
    case 5: if(dcnt==0)
    printf("\nNo Directory's ");
    else
    {
       printf("\nDirectory\tFiles");
       for(i=0;i<dcnt;i++)
       {
          printf("\n%s\t\t",dir[i].dname);
          for(k=0;k<dir[i].fcnt;k++)
          printf("\t%s",dir[i].fname[k]);
       }
     }
    break;

    default:exit(0);
    }
    }

    getch();
    }
```

OUTPUT:

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR1
Directory created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR2

Directory created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A1
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A2
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2
Enter name of the directory – DIR2
Enter name of the file -- B1
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 5
Directory Files
DIR1 A1 A2
DIR2 B1

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 4

Enter name of the directory – DIR
Directory not found

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 3
Enter name of the directory – DIR1
Enter name of the file -- A2

File A2 is deleted

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice – 6

9. Develop a C program to simulate the Linked file allocation strategies.

ALGORITHM:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack.

Step 5: When the page fault occurs replace page present at the bottom of the stack

Step 6: Stop the allocation.

CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
        int f[50], p,i, st, len, j, c, k, a;
        clrscr();

        for (i=0;i<50;i++)
                f[i]=0;
                printf("Enter how many blocks already allocated: ");
                scanf("%d",&p);
                printf("Enter blocks already allocated: ");
        for(i=0;i<p;i++)
        {
                scanf("%d",&a);
                f[a]=1;
        }
        x: printf("Enter index starting block and length: ");
        scanf("%d%d", &st,&len);
        k=len;
        if(f[st]==0)
        {
                for(j=st;j<(st+k);j++)
                {
                        if(f[j]==0)
                        {
                                f[j]=1;
                                printf("%d-------->%d\n",j,f[j]);
                        }

                        else
                        {
                                printf("%d Block is already allocated \n",j);
                                k++;
```

```
                        }
                }
        }
        else
                printf("%d starting block is already allocated \n",st);
                printf("Do you want to enter more file(Yes - 1/No - 0)");
                scanf("%d", &c);
        if(c==1)
                goto x;
                else
                exit(0);
                getch();
}
```

OUTPUT:

Enter how many blocks already allocated: 3

Enter blocks already allocated: 1 3 5

Enter index starting block and length: 2 2

2-------->1

3 Block is already allocated

4-------->1

Do you want to enter more file(Yes - 1/No - 0)0

10. Develop a C program to simulate SCAN disk scheduling algorithm.

ALGORITHM:

In this algorithm, the head starts to scan all the requests in a direction and reaches the end of the disk. After that, it reverses its direction and starts to scan again the requests in its path and serves them. Due to this feature, this algorithm is also known as the "Elevator Algorithm".

CODE:

```
#include <stdio.h>
int request[50];
int SIZE;
int pre;
int head;
int uptrack;
int downtrack;

struct max
{
   int up;
   int down;
}
kate[50];
int dist(int a, int b)
{
   if (a > b)
      return a - b;
   return b - a;
}
void sort(int n)
{
   int i, j;

   for (i = 0; i < n - 1; i++)
  {
      for (j = 0; j < n - i - 1; j++)
     {
         if (request[j] > request[j + 1])
        {
            int temp = request[j];
            request[j] = request[j + 1];
            request[j + 1] = temp;
         }
      }
   }
   j = 0;
   i = 0;
```

```
    while (request[i] != head)
   {
       kate[j].down = request[i];
       j++;
       i++;
    }
    downtrack = j;
    i++;
    j = 0;
    while (i < n)
    {
       kate[j].up = request[i];
       j++;
       i++;
    }
    uptrack = j;
}
void scan(int n)
{
    int i;
    int seekcount = 0;
    printf("SEEK SEQUENCE = ");
    sort(n);
    if (pre < head)
  {
       for (i = 0; i < uptrack; i++)
      {
           printf("%d ", head);
           seekcount = seekcount + dist(head, kate[i].up);
           head = kate[i].up;
       }
       for (i = downtrack - 1; i > 0; i--)
      {
           printf("%d ", head);
           seekcount = seekcount + dist(head, kate[i].down);
           head = kate[i].down;
       }
    }
    else
   {
       for (i = downtrack - 1; i >= 0; i--)
      {
           printf("%d ", head);
           seekcount = seekcount + dist(head, kate[i].down);
           head = kate[i].down;
       }

       for (i = 0; i < uptrack - 1; i++)
      {
           printf("%d ", head);
           seekcount = seekcount + dist(head, kate[i].up);
```

```
            head = kate[i].up;
        }
    }
    printf(" %d\nTOTAL DISTANCE :%d", head, seekcount);
}

int main()
{
    int n, i;

    printf("ENTER THE DISK SIZE :\n");
    scanf("%d", &SIZE);
    printf("ENTER THE NO OF REQUEST SEQUENCE :\n");
    scanf("%d", &n);
    printf("ENTER THE REQUEST SEQUENCE :\n");

    for (i = 0; i < n; i++)
          scanf("%d", &request[i]);
          printf("ENTER THE CURRENT HEAD :\n");
         scanf("%d", &head);
         request[n] = head;
         request[n + 1] = SIZE - 1;
         request[n + 2] = 0;
        printf("ENTER THE PRE REQUEST :\n");
        scanf("%d", &pre);
        scan(n + 3);
}
```

OUTPUT:

Enter the disk size : 4

Enter the no of request sequence :2

Enter the request sequence :

1

2

Enter the current head :1

Enter the pre request : 2

Seek sequence = 1 0 1 2

Total distance :3

CONTENT BEYOND SYLLABUS

1. Write a Shell program to check the given number is even or odd

ALGORITHM:

- Read a number which will be given by user.
- Use `expr $n % 2`. If it equal to 0 then it is even otherwise it is odd.
- Use if-else statements to make this program more easier.
- Must include 'fi' after written 'if-else' statements.

CODE:

```
# HOW TO FIND A NUMBER IS EVEN OR ODD IN SHELL SCRIPT
clear
echo "---- EVEN OR ODD IN SHELL SCRIPT -----"
echo -n "Enter a number:"
read n
echo -n "RESULT: "
if [ `expr $n % 2` == 0 ]
then
        echo "$n is even"
else
        echo "$n is Odd"
    fi
```

OUTPUT:

---- EVEN OR ODD IN SHELL SCRIPT -----"

Enter a number:23

Result : 23 is odd

2. Write a Shell program to check the given year is leap year or not

ALGORITHM:

STEP 1: START SHELL SCRIPT

STEP 2: CLEAR THE SCREEN.

STEP 3: TAKE A YEAR INPUT BY THE USER.

STEP 4: IMPLEMENT IF-ELSE STATEMENT.

STEP 5: GIVE CONDITION #YEAR % 4

STEP 6: IF RESULT ==0 THEN IT IS LEAP YEAR OTHERWISER IT IS NOT

STEP 7: STOP SHELL SCRIPT

CODE:

```
leap=$(date +"%Y")
echo taking year as $leap
if [ `expr $leap % 400` -eq 0 ]
then
echo leap year
elif [ `expr $leap % 100` -eq 0 ]
then
echo not a leap year
elif [ `expr $leap % 4` -eq 0 ]
then
echo leap year
else
echo not a leap year
fi
```

OUTPUT:

$ taking year as 2019

$ not a leap year

3. Write a Shell program to find the factorial of a number

ALGORITHM:

1. Get a number
2. Use for loop or while loop to compute the factorial by using the below formula
3. fact(n) = n * n-1 * n-2 * .. 1
4. Display the result.

CODE:

```
#shell script for factorial of a number
#factorial using while loop
echo"Enter a number"
read num
fact=1
while [ $num -gt 1 ]
do
  fact=$((fact * num))  #fact = fact * num
  num=$((num - 1))      #num = num - 1
done
echo $fact
```

OUTPUT:

```
Enter a number:
    3
    6
Enter a number:
    4
    24
```

4. Write a Shell program to swap the two integers.

ALGORITHM:

1. Store the value of the first number into a temp variable.
2. Store the value of the second number in the first number.
3. Store the value of temp into the second variable.

CODE:

```
# Static input of the
# numbers
first=5
second=10

temp=$first
first=$second
second=$temp

echo "After swapping, numbers are:"
echo "first = $first, second = $second"
```

OUTPUT:

```
After swapping, numbers are:
first = 10, second = 5
```

VIVA QUESTION

1) What is an operating system?

The operating system is a software program that facilitates computer hardware to communicate and operate with the computer software. It is the most important part of a computer system without it computer is just like a box.

2) What is kernel?

Kernel is the core and most important part of a computer operating system which provides basic services for all parts of the OS.

3) What is the use of paging in operating system?

Paging is used to solve the external fragmentation problem in operating system. This technique ensures that the data you need is available as quickly as possible.

4) What is the concept of demand paging?

Demand paging specifies that if an area of memory is not currently being used, it is swapped to disk to make room for an application's need.

5) What are the four necessary and sufficient conditions behind the deadlock?

These are the 4 conditions:

1) **Mutual Exclusion Condition:** It specifies that the resources involved are non-sharable.

2) **Hold and Wait Condition:** It specifies that there must be a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

3) **No-Preemptive Condition:** Resources cannot be taken away while they are being used by processes.

4) **Circular Wait Condition:** It is an explanation of the second condition. It specifies that the processes in the system form a circular list or a chain where each process in the chain is waiting for a resource held by next process in the chain.

6) What is FCFS?

FCFS stands for First Come, First Served. It is a type of scheduling algorithm. In this scheme, if a process requests the CPU first, it is allocated to the CPU first. Its implementation is managed by a FIFO queue.

7) What is deadlock?

Deadlock is a specific situation or condition where two processes are waiting for each other to complete so that they can start. But this situation causes hang for both of them.

8) What is Banker's algorithm?

Banker's algorithm is used to avoid deadlock. It is the one of deadlock-avoidance method. It is named as Banker's algorithm on the banking system where bank never allocates available cash in such a manner that it can no longer satisfy the requirements of all of its customers.

9) What is Pre Emptive Process Scheduling in Operating Systems?

In this instance of Pre Emptive Process Scheduling, the OS allots the resources to a process for a predetermined period of time. The process transitions from running state to ready state or from waiting state to ready state during resource allocation. This switching happens because the CPU may assign other processes precedence and substitute the currently active process for the higher priority process.

10) What is Non Pre Emptive Process Scheduling in Operating Systems?

In this case of Non Pre Emptive Process Scheduling, the resource cannot be withdrawn from a process before the process has finished running. When a running process finishes and transitions to the waiting state, resources are switched.

11) What is paging in Operating Systems?

Paging is a storage mechanism. Paging is used to retrieve processes from secondary memory to primary memory. The main memory is divided into small blocks called pages. Now, each of the pages contains the process which is retrieved into main memory and it is stored in one frame of memory.It is very important to have pages and frames which are of equal sizes which are very useful for mapping and complete utilization of memory.

11) What are Process Scheduling Algorithms in Operating System?

The Process Scheduling Algorithms in Operating Systems are:

1. First Come First Serve CPU Scheduling Algorithm
2. Priority Scheduling CPU Scheduling Algorithm
3. Shortest Job First CPU Scheduling Algorithm
4. Round Robin CPU Scheduling Algorithm
5. Longest Job First CPU Scheduling Algorithm
6. Shortest Remaining Time First CPU Scheduling Algorithm
7. Multiple Queue CPU Scheduling Algorithm

12) What is Round Robin CPU Scheduling Algorithm?

Round Robin is a CPU scheduling mechanism whose cycles around assigning each task a specific time slot. It is the First come First Serve CPU Scheduling method prior Pre Emptive Process Scheduling approach. The Round Robin CPU algorithm frequently emphasizes the Time Sharing method.

13) What is Disk Scheduling in Operating Systems?

Operating systems use disk scheduling to plan when Input or Output requests for the disk will arrive. Input or Output scheduling is another name for disk scheduling.

14) Explain first fit memory allocation technique

In this allocation technique, the process traverses the whole memory and always search for the largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search the largest hole

15) Give the different system call in windows and linux.

Types of System Calls	**Windows**	**Linux**
Process Control	ExitProcess()	exit()
	CreateProcess()	fork()

Types of System Calls	Windows	Linux
	WaitForSingleObject()	wait()
File Management	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
	SetConsoleMode()	ioctl()
Device Management	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	Sleep()	sleep()
	CreatePipe()	pipe()

15) What is turnaround time?

Turnaround time is the interval from the time of submission to the time of completion of a process. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/0 operations.