18AIC208J - OPERATING SYSTEM DESIGN LABORATORY

(2018 Regulation)

II Year/ III Semester

Academic Year: 2022 -2023

By

CHARVI JAIN (RA2111047010113)

**DEPARTMENT OF
COMPUTATIONAL INTELLIGENCE**

**FACULTY OF ENGINEERING AND TECHNOLOGY
SCHOOL OF COMPUTING
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
Kattankulathur, Chengalpattu**

BONAFIDE

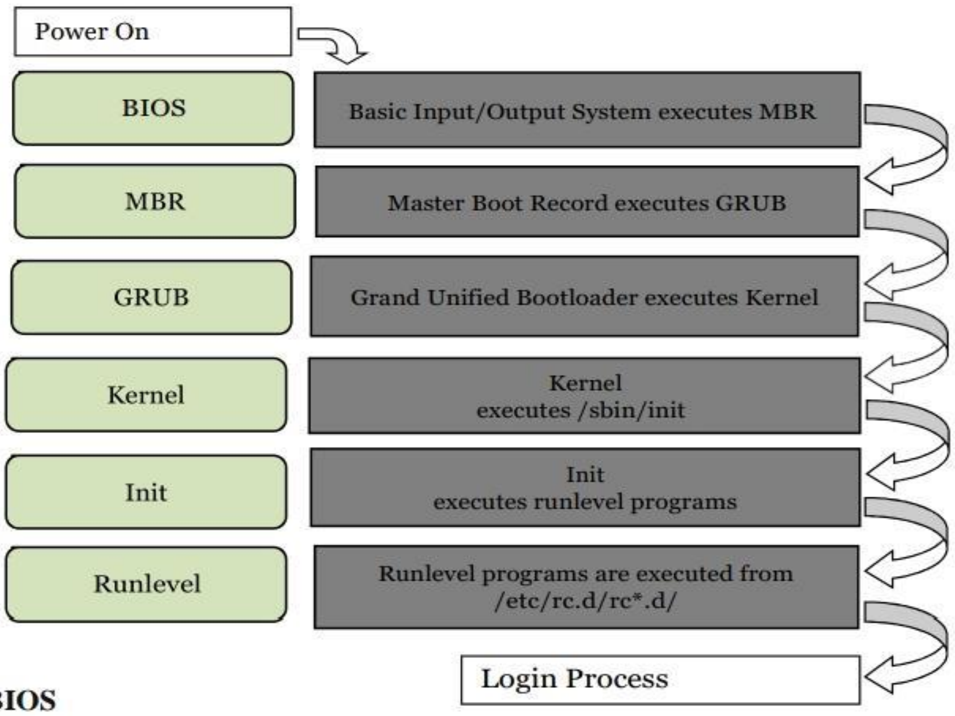
This is to certify that **18AIC208J - OPERATING SYSTEM DESIGN LABORATORY Record** is the bonafide work of **CHARVI JAIN (RA2111047010113)** who completed the record within the allotted time.

Signature of the Faculty

Dr. T. Subha	Dr. R. Annie Uthra
Assistant Professor	**Professor and Head**
Department of CINTEL,	Department of CINTEL,
SRM Institute of Science and Technology	SRM Institute of Science and Technology

EXPERIMENT-1
BOOTING PROCESS OF LINUX

Press the power button on your system, and after few moments you see the Linux login prompt. From the time you press the power button until the Linux login prompt appears, the following sequence occurs. The following are the 6 high level stages of a typical Linux boot process.



Power On
BIOS
Basic Input/Output System executes MBR
MBR
Master Boot Record executes GRUB
GRUB
Grand Unified Bootloader executes Kernel
Kernel
Kernel executes /sbin/init
Init
Init executes runlevel programs
Runlevel
Runlevel programs are executed from /etc/rc.d/rc*.d/
Login Process

Step 1.BIOS

- BIOS stands for Basic Input/Output System
- Performs some system integrity checks
- Searches, loads, and executes the boot loader program.
- It looks for boot loader in floppy, CD-ROMs, or hard drive. You can press a key (typically F12 or F2, but it depends on your system) during the BIOS startup to change the boot sequence.
- Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
- So, in simple terms BIOS loads and executes the MBR boot loader.

Step 2. MBR

- MBR stands for Master Boot Record.
- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
- MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
- It contains information about GRUB (or LILO in old systems).
- So, in simple terms MBR loads and executes the GRUB boot loader.

Step 3. GRUB

- GRUB stands for Grand Unified Bootloader.
- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen, waits for few seconds, if you don't enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn't understand filesystem).
- Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.

```
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title CentOS(2.6.18-194.el5PAE)
root(hd0,0)
kernel/boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/
initrd /boot/initrd-2.6.18-194.el5PAE.img
```

- As you notice from the above info, it contains kernel and initrd image.
- So, in simple terms GRUB just loads and executes Kernel and initrd images.

Step 4. Kernel

- Mounts the root file system as specified in the "root=" in grub.conf
- Kernel executes the /sbin/init program
- Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a 'ps -ef | grep init' and check the pid.
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

Step 5. Init

- ➢ Looks at the /etc/inittab file to decide the Linux run level.
- ➢ Following are the available run levels

  0 – halt
  1 – Single user mode
  2 – Multiuser, without NFS
  3 – Full multiuser mode
  4 – unused
  5 – X11
  6 – reboot
- ➢ Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.

- Execute 'grep initdefault /etc/inittab' on your system to identify the default run level
- If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
- Typically you would set the default run level to either 3 or 5.

Step 6. Runlevel programs

- When the Linux system is booting up, you might see various services getting started. For example, it might say "starting sendmail …. OK". Those are the runlevel programs, executed from the run level directory as defined by your run level.
- Depending on your default init level setting, the system will execute the programs from one of the following directories.

  Run level 0 – /etc/rc.d/rc0.d/
  Run level 1 – /etc/rc.d/rc1.d/
  Run level 2 – /etc/rc.d/rc2.d/
  Run level 3 – /etc/rc.d/rc3.d/
  Run level 4 – /etc/rc.d/rc4.d/
  Run level 5 – /etc/rc.d/rc5.d/
  Run level 6 – /etc/rc.d/rc6.d/
- Please note that there are also symbolic links available for these directory under /etc directly. So, /etc/rc0.d is linked to /etc/rc.d/rc0.d.
- Under the /etc/rc.d/rc*.d/ directories, you would see programs that start with S and K.
- Programs starts with S are used during startup. S for startup.
- Programs starts with K are used during shutdown. K for kill.
- There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
- For example, S12syslog is to start the syslog deamon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.

Login Process

1. Users enter their username and password
2. The operating system confirms your name and password.
3. A "shell" is created for you based on your entry in the "/etc/passwd" file
4. You are "placed" in your "home"directory.
5. Start-up information is read from the file named "/etc/profile". This file is known as the system login file. When every user logs in, they read the information in this file.
6. Additional information is read from the file named ".profile" that is located in your "home" directory. This file is known as your personal login file.

RESULT:

The booting process of LINUX is seen successfully.

EXPERIMENT-2
BASIC LINUX COMMANDS

AIM:

To execute simple Linux commands in VI editor.

COMMANDS:

1. echo AI →To display the string AI.
2. clear → To clear the screen.
3. date →To display the current date and time.
4. cal2003 → To display the calendar of the year 2003.
5. ls → List files in the present working directory.
6. cat > temp1 → To create the file temp1.
7. cat temp1 → Display the content in the file temp1.
8. who → Display the no. of user accounts.
9. who am I → Display the user working.
10. wc temp → List of no of characters, words & lines of the file temp.
11. cp temp1 t1 → Copy file temp1 into t1.
12. mv temp1 t1 → Rename file temp1 to t1.
13. rm t1 → Remove the file t1.
14. head -5 t1 → List first 5 lines of the file t1.
15. tail -5 t1 → List last 5 lines of the file t1.
16. mkdir f3 → Creates a directory f3.
17. cd f1 → Changes to the directory t1.

RESULT:

The above commands were executed successfully in the in VI editor.

EXPERIMENT-3
PROGRAMS USING FORK

Q1. Find the output of the following program

AIM:

To write a program using fork command.

ALGORITHM:

```
Step 1: Start.
Step 2: Initialize a and b.
Step 3: Print the numbers.
Step 4: Initialize a new variable pid and call fork () function to it.
Step 5: Check whether pid=0 or not, If pid=0, print the variables by incrementing them
by 1. Else print them by decrementing 1.
Step 6: Stop.
```

CODE:

```
#include <stdio.h>
#include<unistd.h>
int main()
{   int a=5,b=10,pid;
    printf("Before fork a=%d b=%d \n",a,b);
    pid=fork();
    if(pid==0)
    {   a=a+1;
        b=b+1;
        printf("In child a=%d b=%d \n",a,b); }
    else
    {   sleep(1);
        a=a-1;
        b=b-1;
        printf("In Parent a=%d b=%d \n",a,b); }
    return 0; }
```

OUTPUT:

```
Before fork a=5 b=10
In child a=6 b=11
In Parent a=4 b=9
```

RESULT:

The above program was executed successfully.

Q2. Calculate the number of times the text “SRMIST” is printed.

AIM:

To use the command, fork() multiple times and calculate the number of times the text “SRMIST” is printed.

ALGORITHM:

Step-1: Start.
Step-2: Call the function fork() three times.
Step-3: Print SRMIST.
Step-4: Stop.

CODE:

```
#include <stdio.h>
#include<unistd.h>
int main()
{    fork();
     fork();
     fork();
     printf(“SRMIST\n”);
     return 0;
}
```

OUTPUT:

```
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
```

RESULT:

The above program was executed and verified successfully.

Q3. Complete the following program as described below:
The child process calculates the sum of odd numbers and the parent process calculates the sum of even numbers up to the number ‘n’. Ensure the Parent process waits for the child process to finish.

AIM:

To calculate the Sum of odd numbers with the child process and the sum of even numbers with the parent process.

ALGORITHM:

Step 1: Start.
Step 2: Initialize the variables i, pid, n, oddsum and evensum.
Step 3: Get n, as the limit from the user.
Step 4: Using a for loop, add odd numbers in the child process and even numbers in the parent process.
Step 5: Print the sum of odd and even numbers respectively as output.
Step-6: Stop.

CODE:

```
#include <stdio.h>
#include <unistd.h>
int main()
{   int i, pid, n, oddsum=0, evensum=0;
    printf("Enter the value of n: ");
    scanf("%d", &n);
    pid = fork();
    for (i=0; i<=n;i++)
    {   if (pid==0)
        {   if (i%2==1)
            {   oddsum = oddsum+i;
            }
            else
            {   evensum=evensum +i;
            }
        }
        else
        {   wait(1);
            evensum= evensum+i;
        }
    }
    printf ("Sum of odd numbers: %d\n",oddsum);
    printf ("Sum of even numbers: %d\n",evensum);
    return 0;
}
```

OUTPUT:

```
Enter the value of n: 10
```

```
Sum of odd numbers: 25
Sum of even numbers: 30
```

RESULT:

The sum of odd numbers is done by the child process and the sum of even numbers is done by the parent process. This program was executed successfully.

Q4. How many child processes are created for the following code?

AIM:

To find the number of child processes created in the following code.

ALGORITHM:

Step 1: Start.
Step 2: Call multiple fork() system calls and implement any logical operation on it.
Step 3: Print a string to check how many child processes have been created.
Step 4: Stop.

CODE :

```
#include <stdio.h>
#include<unistd.h>
int main()
{   fork();
    fork()&&fork()||fork();
    fork();
    printf(“Yes ”);
    return 0;
}
```

OUTPUT:

```
Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes Yes
Yes
```

RESULT:

The number of child processes have been calculated successfully.

Q5. How many child processes are created for the following code?

AIM:

To find the number of child processes created in the following code.

ALGORITHM:

Step 1: Start.
Step 2: Initialize the variable n.
Step 3: Use a for loop to call fork() function 'n' number of times.
Step 4: Print a string to check how many times the child processes have been created.
Step 5: Stop.

CODE:

```
#include <stdio.h>
#include<unistd.h>
int main()
{  int n=2,i;
    for(i=0; i<n;i++)
    {  fork();
        printf("SRMIST\n");
    }
    return 0;
}
```

OUTPUT:

```
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
```

RESULT:

The number of child processes have been calculated successfully.

Q6. Write a program to display the parent and child id.

AIM:

To write a program to display the parent and child ID

ALGORITHM:

```
Step 1: Start.
Step 2: Declare the variables p_id, c_pid.
Step 3: Call getpid() to get the id of the child process and getppid() to get the parent
process id and store it in variables respectively.
Step 4: Print the process id’s of both child and parent processes.
Step 5: Stop.
```

CODE:

```
#include <stdio.h>
#include <unistd.h>
int main()
{   int c_id, p_pid;
    c_id = getpid();
    p_pid = getppid();
    printf("Child Process ID: %d\n", c_id);
    printf("Parent Process ID: %d\n", p_pid);
    return 0;
}
```

OUTPUT:

```
Child Process ID: 8956
Parent Process ID: 8946
```

RESULT:

A program to display the parent and child process ID was executed successfully.

EXPERIMENT-4
PRODUCER CONSUMER USING SEMAPHORES

AIM:

To simulate producer-consumer using Semaphores.

ALGORITHM:

Step 1: Start.
Step 2: Create int variables mutex=1, full=0, empty=0, x=0.
Step 3: Create a wait function with a parameter that increments the parameter.
Step 4: Create a signal function with a parameter that increments the parameter.
Step 5: Create a producer function that calls wait with mutex and assigns it to mutex, calls signal with full and assigns it to full, calls wait with empty and assigns it to empty. Increment the variable x and print produce produced product x. Then call the signal with mutex and assign it to mutex.
Step 6: Create a consumer function that calls wait with mutex and assigns it to mutex, calls wait with full and assigns it to full, calls signal with empty and assigns it to empty. decrement the variable x and print produce produced product x. Then call the signal with mutex and assign it to mutex.
Step 7: Get the user choice.
Step 8: If the choice is 1 then call the producer function. Else if choice is 2 then call the customer function. Else call the exit function.
Step 9: Stop.

CODE:

```
#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:2
Consumer consumes item 1
Enter your choice:1
Producer produces the item 1
Enter your choice:2
Consumer consumes item 1
```

```
Enter your choice:2
Buffer is empty!!
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
```

RESULT:

Producer-consumer using Semaphores were executed successfully.

EXPERIMENT-5
PIPES

EXPERIMENT-5.1:

AIM:

To simulate pipe technique.

ALGORITHM:

```
Step 1: Start.
Step 2: Create character data typed array's p of size 2 and buff of size 25.
Step 3: Create a pipe for the array p.
Step 4: Check if the fork()=0,
        If true then print, child is writing and use write function to write into the pipe.
        If false, then print parent is reading and use the read function to read the pipe
        content and store it in buff.
Step 5: Print buff.
Step 6: Stop.
```

CODE:

```
#include <stdio.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{   int p[2];
    char buff[25];
    pipe(p);
    if(fork()==0)
    {   printf("Child : Writing to pipe \n");
        write(p[1],"Welcome",8);
    printf("Child Exiting\n");
    }
    else
    {   wait(NULL);
        printf("Parent : Reading from pipe \n");
        read(p[0],buff,8);
        printf("Pipe content is : %s \n",buff);
    }
    return 0;
}
```

OUTPUT:

```
Child: writing to pipe
```

```
Child Exiting
Parent: Reading from pipe
Pipe content is: Welcome
```

RESULT:

The above program was executed and verified successfully.

EXPERIMENT-5.2:

AIM:

To simulate pipe technique using an external file.

ALGORITHM:

```
Step 1: Start.
Step 2: Create a character data typed array buff of size 25 and int variable rfd and wfd.
Step 3: Check if the fork()=0,
        If true then print, child is writing and use open function to open the file in
        O_WRONLY mode and assign it to wfd. Use the write function to write into the
        pipe.
        If false, then print parent is reading and use open function to open the file in
        O_RDONLY mode and assign it to rfd. Use the read function to read the pipe
        content and store it in buff.
Step 5: Print buff.
Step 6: Stop.
```

CODE:

```
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
int main ()
{  char buff[25]; int rfd,wfd;
   mkfifo("fif1", O_CREAT|0644);
   if(fork ()==0)
   {  printf("Child writing into FiF1\n");
      wfd=open("fif1", O_WRONLY);
      write(wfd,"Hello",6);}
   else
   {  rfd=open("fif1",O_RDONLY);
      read(rfd, buff,6);
      printf("Parent reads from FiF1 : %s\n",buff); }
   return 0;}
```

OUTPUT:

```
Parent reads from FiF1: Hello
Child writing into FiF1
```

RESULT:

The above program was executed and verified successfully.

EXPERIMENT - 6
CPU SCHEDULING

FIRST COME FIRST SERVE CPU SCHEDULING:

AIM:

To simulate First Come First Serve CPU scheduling algorithm.

ALGORITHM:

```
Step 1 : Input the number of processes required to be scheduled using FCFS, burst
         time for each process and its arrival time.
Step 2 : Using enhanced bubble sort technique, sort the all given processes in
         ascending order according to arrival time in a ready queue.
Step 3 : Calculate the Finish Time, Turn Around Time and Waiting Time for each
         process which in turn help to calculate Average Waiting Time and Average
         Turnaround Time required by CPU to schedule a given set of processes using
         FCFS.
Step 3.1 : for i = 0, Finish Time T 0 = Arrival Time T 0 + Burst Time T 0
Step 3.2 : for i >= 1, Finish Time T i = Burst Time T i + Finish Time T i - 1
Step 3.3 : for i = 0, Turn Around Time T 0 = Finish Time T 0 - Arrival Time T 0
Step 3.4 : for i >= 1, Turn Around Time T i = Finish Time T i - Arrival Time T i
Step 3.5 : for i = 0, Waiting Time T 0 = Turn Around Time T 0 - Burst Time T 0
Step 3.6 : for i >= 1, Waiting Time T i = Turn Around Time T i - Burst Time T i - 1
Step 4 : Process with less arrival time comes first and gets scheduled first by the CPU.
Step 5 : Calculate the Average Waiting Time and Average Turnaround Time.
Step 6 : Stop.
```

CODE:

```c
#include <stdio.h>
int main()
{   int n,i,fsfc_waittime=0,process_no=1,execute_time;
    float toe_avg=0,wt_avg=0;
    printf("Enter the total no.of processes:");
    scanf("%d",&n);
    int burst_time[n];
    for (i=0;i<n;i++)
    {   printf("Enter the Process-%d burst time:",process_no);
        scanf("%d",&burst_time[i]);
        process_no++;
    }
    process_no=1;
    printf("      Process Queue    Burst time   Wait time   TurnAround time\n");
    for(i=0;i<n;i++)
    {   execute_time+=burst_time[i];
        printf("      P%d                   %d          %d
%d\n",process_no,burst_time[i],fsfc_waittime,execute_time);
        wt_avg+=fsfc_waittime;
        toe_avg+=burst_time[i];
        if (i<n-1)
        fsfc_waittime+=burst_time[i];
        process_no++;
    }
    return 0;
}
```

OUTPUT:

```
Enter the total no.of processes:5
Enter the Process-1 burst time:6
Enter the Process-2 burst time:2
Enter the Process-3 burst time:8
Enter the Process-4 burst time:3
Enter the Process-5 burst time:4
Process Queue    Burst time   Wait time
     P1               6           0
     P2               2           6
     P3               8           8
     P4               3           16
     P5               4           19
```

RESULT:

The First Come First Serve (FCFS) CPU scheduling was simulated successfully.

SHORTEST JOB FIRST CPU SCHEDULING:

AIM:

To simulate Shortest Job First CPU scheduling algorithm.

ALGORITHM:

Step 1: Start.
Step 2: Sort all the processes according to their arrival time.
Step 3: Select the process with minimum arrival time as well as minimum burst time.
Step 4: After completion of the process, select from the ready queue the process which has the minimum burst time.
Step 5: Repeat above processes until all processes have finished their execution.
Step 6: Stop.

CODE:

```
#include <stdio.h>
int main()
{   int A[100][4];
    int i, j, n, total = 0, index, temp;
    float avg_wt, avg_tat;
    printf("Enter number of process: ");
    scanf("%d", &n);
    printf("Enter Burst Time:\n");
    for (i = 0; i < n; i++)
    {   printf("P%d: ", i + 1);
        scanf("%d", &A[i][1]);
        A[i][0] = i + 1;
    }
    for (i = 0; i < n; i++)
    {   index = i;
        for (j = i + 1; j < n; j++)
            if (A[j][1] < A[index][1])
                index = j;
                temp = A[i][1];
                A[i][1] = A[index][1];
                A[index][1] = temp;
                temp = A[i][0];
                A[i][0] = A[index][0];
                A[index][0] = temp;
    }
    A[0][2] = 0;
    for (i = 1; i < n; i++)
    {   A[i][2] = 0;
        for (j = 0; j < i; j++)
            A[i][2] += A[j][1];
        total += A[i][2];
    }
    avg_wt = (float)total / n;
    total = 0;
    printf("P     BT     WT     TAT\n");
    for (i = 0; i < n; i++)
    {   A[i][3] = A[i][1] + A[i][2];
        total += A[i][3];
        printf("P%d     %d     %d      %d\n", A[i][0], A[i][1], A[i][2], A[i][3]);
    }
    avg_tat = (float)total / n;
    printf("Average Waiting Time= %f", avg_wt);
    printf("\nAverage Turnaround Time= %f", avg_tat);
}
```

OUTPUT:

```
Enter number of process: 5
Enter Burst Time:
P1: 6
P2: 2
P3: 8
P4: 3
P5: 4
P     BT     WT     TAT
P2     2      0       2
P4     3      2       5
P5     4      5       9
P1     6      9       15
P3     8      15      23
Average Waiting Time= 6.200000
Average Turnaround Time= 10.800000
```

RESULT:

The Shortest Job First (SJF) CPU scheduling was simulated successfully.

PRIORITY CPU SCHEDULING:

AIM:

To simulate Priority CPU scheduling algorithm.

ALGORITHM:

```
Step 1: Start.
Step 2: Input the number of processes.
Step 3: Input the arrival time and burst time.
Step 4: Sort the element on the basis of priority.
Step 5: Calculate the wait time and turnaround time for all the processes.
Step 6: Calculate the average wait time and average turnaround time.
Step 7: Display them.
Step 8: Stop.
```

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 b[n],p[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",&b[i],&p[i]);
        index[i]=i+1;
    }
    for(int i=0;i<n;i++)
    {   int a=p[i],m=i;
        for(int j=i;j<n;j++)
        {   if(p[j] > a)
            {   a=p[j];
                m=j;
            }
        }
        swap(&p[i], &p[m]);
        swap(&b[i], &b[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+b[i]);
        t+=b[i];
    }
    printf("\n");
    printf("Process Id     Burst Time   Wait Time    TurnAround Time\n");
    int wait_time=0;
    for(int i=0;i<n;i++)
    {   printf("P%d          %d           %d           %d\n",index[i],b[i],wait_time,wait_time + b[i]);
        wait_time += b[i];
    }
    return 0;
}
```

OUTPUT:

```
Enter Number of Processes: 3
Enter Burst Time and Priority Value for Process 1: 2 0
Enter Burst Time and Priority Value for Process 2: 4 2
Enter Burst Time and Priority Value for Process 3: 7 1
Order of process Execution is
P2 is executed from 0 to 4
P3 is executed from 4 to 11
P1 is executed from 11 to 13

Process Id     Burst Time   Wait Time    TurnAround Time
P2          4          0          4
P3          7          4          11
P1          2          11           13
```

RESULT:

The Priority CPU scheduling was simulated successfully.

SHORTEST REMAINING TIME FIRST CPU SCHEDULING:

AIM:

To simulate Shortest Remaining Time First CPU scheduling algorithm.

ALGORITHM:

Step 1: Start.
Step 2: Input the number of processes.
Step 3: Input the arrival time and burst time.
Step 4: Calculate the wait time and turnaround time for all the processes.
Step 5: Calculate the average wait time and average turnaround time.
Step 6: Display them.
Step 7: Stop.

CODE:

```
#include <stdio.h>
int main()
{   int n, ari[10], bur[10], total = 0, i, j, small, temp, procs[100], k, waiting[10], finish[10];
    float tavg = 0.0, wavg = 0.0;
    printf("ENTER THE NUMBER OF PROCESSES:");
    scanf("%d", & n);
    for (i = 0; i < n; i++)
    {   printf("ENTER THE ARRIVAL TIME OF PROCESS %d:\t", i);
        scanf("%d", & ari[i]);
        printf("ENTER THE BURST TIME OF PROCESS %d:\t", i);
        scanf("%d", & bur[i]);
        waiting[i] = 0;
        total += bur[i];
    }
    for (i = 0; i < n; i++)
    {   for (j = i + 1; j < n; j++)
        {   if (ari[i] > ari[j])
            {   temp = ari[i];
                ari[i] = ari[j];
                ari[j] = temp;
                temp = bur[i];
                bur[i] = bur[j];
                bur[j] = temp;
            }
        }
    }
    for (i = 0; i < total; i++)
    {   small = 3200;
        for (j = 0; j < n; j++)
        {   if ((bur[j] != 0) && (ari[j] <= i) && (bur[j] < small))
            {   small = bur[j];
                k = j;
            }
        }
        bur[k]--;
        procs[i] = k;
    }
    k = 0;
    for (i = 0; i < total; i++)
    {   for (j = 0; j < n; j++)
        {   if (procs[i] == j)
            {   finish[j] = i;
                waiting[j]++;
            }
        }
    }
    for (i = 0; i < n; i++)
    {   printf("\n PROCESS %d:-FINISH TIME==> %d TURNAROUND TIME==>%d
WAITING TIME==>%d\n", i + 1, finish[i] + 1, (finish[i] - ari[i]) + 1, (((finish[i] + 1) -
waiting[i]) - ari[i]));
        wavg = wavg + (((finish[i] + 1) - waiting[i]) - ari[i]);
        tavg = tavg + ((finish[i] - ari[i]) + 1);
    }
    printf("\n WAvG==>\t%f\n TAVG==>\t%f\n", (wavg / n), (tavg / n));
    return 0;
}
```

OUTPUT:

```
ENTER THE NUMBER OF PROCESSES:5
ENTER THE ARRIVAL TIME OF PROCESS 0:      2
ENTER THE BURST TIME OF PROCESS 0:        6
ENTER THE ARRIVAL TIME OF PROCESS 1:      5
ENTER THE BURST TIME OF PROCESS 1:        2
ENTER THE ARRIVAL TIME OF PROCESS 2:      1
ENTER THE BURST TIME OF PROCESS 2:        8
ENTER THE ARRIVAL TIME OF PROCESS 3:      0
ENTER THE BURST TIME OF PROCESS 3:        3
ENTER THE ARRIVAL TIME OF PROCESS 4:      4
ENTER THE BURST TIME OF PROCESS 4:        4

PROCESS 1: -FINISH TIME==> 3 TURNAROUND TIME==>3 WAITING TIME==>0

PROCESS 2: -FINISH TIME==> 23 TURNAROUND TIME==>22 WAITING TIME==>14

PROCESS 3: -FINISH TIME==> 15 TURNAROUND TIME==>13 WAITING TIME==>7

PROCESS 4: -FINISH TIME==> 10 TURNAROUND TIME==>6 WAITING TIME==>2

PROCESS 5: -FINISH TIME==> 7 TURNAROUND TIME==>2 WAITING TIME==>0

 WAVG==>     4.600000
 TAVG==>     9.200000
```

RESULT:

The Shortest remaining time first was executed successfully.

ROUND ROBIN CPU SCHEDULING:

AIM:

To simulate Round Robin Algorithm.

ALGORITHM:

Step 1: Start.
Step 2: Input the number of processes.
Step 3: Input the arrival time, burst time and time quantum for the processes.
Step 4: Calculate the total of wait time and turnaround time for all the processes.
Step 5: Calculate the average wait time and average turnaround time.
Step 6: Display them.
Step 7: Stop.

CODE:

```
#include<stdio.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;
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");
scanf("%d", &at[i]);
printf(" \nBurst time is: \t");
scanf("%d", &bt[i]);
temp[i] = bt[i];
}
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
printf("\n Process No \t\t Burst Time \t\t Waiting Time \t\t TAT ");
for(sum=0, i = 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0)
{
   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--;
        printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i]-bt[i],sum-
at[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;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
}
```

OUTPUT:

```
Total number of processes in the system: 5
Enter the Arrival and Burst time of the Process [1]
Arrival time is:        0
Burst time is:          6
Enter the Arrival and Burst time of the Process [2]
Arrival time is:        0
Burst time is:          2
Enter the Arrival and Burst time of the Process [3]
Arrival time is:        0
Burst time is:          8
Enter the Arrival and Burst time of the Process [4]
Arrival time is:        0
Burst time is:          3
Enter the Arrival and Burst time of the Process [5]
Arrival time is:        0
Burst time is:          4
Enter the Time Quantum for the process:         4
Process No          Burst Time                  Waiting Time            TAT
Process No [2]          2                    4               6
Process No [4]          3                    10              13
Process No [5]          4                    13              17
Process No [1]          6                    13              19
Process No [3]          8                    15              23
Average Turn Around Time:    11.000000
Average Waiting Time:    15.600000
```

RESULT:

The Round robin CPU scheduling was executed successfully.

EXPERIMENT-7
BANKER'S ALGORITHM

AIM:

To simulate banker's algorithm for deadlock avoidance.

ALGORITHM:

Step 1: Start.
Step 2: Initialize alloc, max, avail, need and ans matrices.
Step 3: Calculate the need matrix by subtracting the alloc matrix from the max matrix.
Step 4: Repeat Step-3 for all the processes.
Step 5: Check whether the need matrix is less than or equal to avail matrix, if it is true add need to avail and print the process.
Step 6: Repeat Step-5 until every process's need matrix is less than the available matrix.
Step 7: Stop.

CODE:

```
#include <stdio.h>
int main()
{   int p, r, i, j, k;
    p = 5;
    r = 3;
    int alloc[5][3] = { { 0, 1, 0 },
                        { 2, 0, 0 },
                        { 3, 0, 2 },
                        { 2, 1, 1 },
                        { 0, 0, 2 } };
    int max[5][3] = { { 7, 5, 3 },
                      { 3, 2, 2 },
                      { 1, 0, 1 },
                      { 2, 2, 2 },
                      { 4, 3, 3 } };
    int avail[3] = { 3, 3, 2 };
    int f[p], ans[p], ind = 0;
    for (k = 0; k < p; k++)
    {   f[k] = 0;
    }
    int need[p][r];
    for (i = 0; i < p; i++)
    {   for (j = 0; j < r; j++)
        {   need[i][j] = max[i][j] - alloc[i][j];
        }
    }
    int y = 0;
    for (k = 0; k < 5; k++)
    {   for (i = 0; i < p; i++)
        {   if (f[i] == 0)
            {   int flag = 0;
                for (j = 0; j < r; j++)
                {   if (need[i][j] > avail[j])
                    {   flag = 1;
                        break;
                    } }
                if (flag == 0)
                {   ans[ind++] = i;
                    for (y = 0; y < r; y++)
                    {   avail[y] += alloc[i][y];
                        f[i] = 1;
                    } }
            }
        }
        int flag = 1;
        for(int i=0;i<p;i++)
        {   if(f[i]==0)
            {   flag=0;
                printf("The following system is not safe");
                break;
            } }
        if(flag==1)
        {   printf("Following is the SAFE Sequence\n");
            for (i = 0; i < p - 1; i++)
            {   printf(" P%d ->", ans[i]);
            }
            printf(" P%d", ans[p - 1]);
        } }
    return (0);
}
```

OUTPUT:

```
Following is the SAFE Sequence
P1 -> P2 -> P3 -> P4 -> P0
```

RESULT:

Banker’s deadlock avoidance algorithm was simulated successfully.

EXPERIMENT-8
PAGING TECHNIQUES

EXPERIMENT-8.1:

AIM:

To simulate paging technique of memory management technique.

ALGORITHM:

Step 1: Start.
Step 2: Initialize int variables ms, ps, nop, np, rempages, pa, offset, x, y and the array’s s, fno.
Step 3: Get ms and ps from the user.
Step 4: Calculate nop by dividing ms by ps and print nop.
Step 5: Get np from the user.
Step 6: Get s from the user and calculate rempages by subtracting s from nop.
Step 7: Get fno from the user if s is less than rempages.
Step 8: Repeat step-6 and step-7 for np times.
Step 9: Get x, y, offset from the user.
Step 10: Check if x is greater than np and offset is greater than ps. If true, calculate pa by multiplying fno value at position x, y with ps and adding it to offset.
Step 11: Print pa.
Step 12: Stop.

CODE:

```
#include<stdio.h>
Int 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);
        }
}
```

OUTPUT:

```
Enter the memory size -- 1000
Enter the page size -- 100
The no. of pages available in memory are -- 10
Enter number of processes -- 3
Enter no. of pages required for p[1]-- 4
Enter pagetable for p[1] --- 8 6 9 5
Enter no. of pages required for p[2]-- 5
Enter pagetable for p[2] --- 1 4 5 7 3
Enter no. of pages required for p[3]-- 5
Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2 3 60
The Physical Address is -- 760
```

RESULT:

The Paging technique of memory management technique was simulated successfully.

EXPERIMENT-8.2:

AIM:

To simulate memory management technique - Paging.

ALGORITHM:

Step-1: Start.
Step-2: Read all the necessary input from the keyboard.
Step-3: Pages - Logical memory is broken into fixed - sized blocks.
Step-4: Frames – Physical memory is broken into fixed – sized blocks.
Step-5: Calculate the physical address using the following
Physical address = (Frame number * Frame size) + offset
Step-7: Display the physical address.
Step-8: Stop.

CODE:

```
#include <stdio.h>
struct pstruct{
        int fno;
        int pbit;
}ptable[10];
int pmsize,lmsize,psize,frame,page,ftable[20],frameno;
void info(){
        printf("\n\nMEMORY MANAGEMENT USING PAGING\n\n");
        printf("\n\nEnter the Size of Physical memory: ");
        scanf("%d",&pmsize);
        printf("\n\nEnter the size of Logical memory: ");
        scanf("%d",&lmsize);
        printf("\n\nEnter the partition size: ");
        scanf("%d",&psize);
        frame = (int) pmsize/psize;
        page = (int) lmsize/psize;
        printf("\nThe physical memory is divided into %d no.of frames\n",frame);
        printf("\nThe Logical memory is divided into %d no.of pages",page);
}
void assign(){
        int i;
        for (i=0;i<page;i++){
                ptable[i].fno = -1;
                ptable[i].pbit= -1;
        }
        for(i=0; i<frame;i++){
                ftable[i] = 32555;
        }
        for (i=0;i<page;i++){
                printf("\n\nEnter the Frame number where page %d must be placed: ",i);
                scanf("%d",&frameno);
                ftable[frameno] = i;
                if(ptable[i].pbit == -1){
                        ptable[i].fno = frameno;
                        ptable[i].pbit = 1;
                }
        }
        printf("\n\nPAGE TABLE\n\n");
        printf("PageAddress FrameNo. PresenceBit\n\n");
        for (i=0;i<page;i++){
                printf("%d\t\t%d\t\t%d\n",i,ptable[i].fno,ptable[i].pbit);
        }
        printf("\n\n\n\tFRAME TABLE\n\n");
        printf("FrameAddress PageNo\n\n");
        for(i=0;i<frame;i++){
                printf("%d\t\t%d\n",i,ftable[i]);
        }
}
void cphyaddr(){
        int laddr,paddr,disp,phyaddr,baddr;
        printf("\n\n\n\tProcess to create the Physical Address\n\n");
        printf("\nEnter the Base Address: ");
        scanf("%d",&baddr);
        printf("\nEnter theLogical Address: ");
        scanf("%d",&laddr);
        paddr = laddr / psize;
        disp = laddr % psize;
        if(ptable[paddr].pbit == 1 ){
                phyaddr = baddr + (ptable[paddr].fno*psize) + disp;
        }
        printf("\nThe Physical Address where the instruction present: %d",phyaddr);
}
void main(){
        info();
        assign();
        cphyaddr();
}
```

OUTPUT:

```
MEMORY MANAGEMENT USING PAGING

Enter the Size of Physical memory: 16
Enter the size of Logical memory: 8
Enter the partition size: 2
The physical memory is divided into 8 no.of frames

The Logical memory is divided into 4 no.of pages

Enter the Frame number where page 0 must be placed: 5
Enter the Frame number where page 1 must be placed: 6
Enter the Frame number where page 2 must be placed: 7
Enter the Frame number where page 3 must be placed: 2

PAGE TABLE

PageAddress FrameNo. PresenceBit

0           5           1
1           6           1
2           7           1
3           2           1

FRAME TABLE

FrameAddress PageNo

0           32555
1           32555
2           3
3           32555
4           32555
5           0
6           1
7           2

Process to create the Physical Address

Enter the Base Address: 1000
Enter theLogical Address: 3
The Physical Address where the instruction present: 1013
```

RESULT:

Memory management technique - Paging was simulated successfully.

EXPERIMENT-8.3:

AIM:

To simulate paging technique of memory management technique.

ALGORITHM:

```
Step-1: Start.
Step-2: Initialize int variables i, f, n, ps, off, pno, choice and the array page.
Step-3: Get n and ps from the user.
Step-4: Get f from the user.
Step-5: Get page from the user.
Step-6: Repeat step-6 for f times.
Step-7: Get pno and off from the user.
Step-8: If the value of page at pno is -1 then print page not available. Else print value of
page at pno and off.
Step-9: Stop.
```

CODE:

```
#include<stdio.h>
#define MAX 50
int main()
{
        int page[MAX],i,n,f,ps,off,pno;
        int choice=0;
        printf("\nEnter the no of pages in memory: ");
        scanf("%d",&n);
        printf("\nEnter page size: ");
        scanf("%d",&ps);
        printf("\nEnter no of frames: ");
        scanf("%d",&f);
        for(i=0;i<n;i++)
        {    page[i]=-1;
        }
        printf("\nEnter the page table\n");
        printf("(Enter frame no as -1 if that page is not present in any frame)\n\n");
        printf("\npageno\tframeno\n-------\t-------");
        for(i=0;i<n;i++){
                printf("\n\n%d\t\t",i);
                scanf("%d",&page[i]);
        }
        do{
                printf("\n\nEnter the logical address(i.e,page no & offset):");
                scanf("%d%d",&pno,&off);
                if(page[pno]==-1){
                        printf("\n\nThe required page is not available in any of frames");
                }
                else
                {
                        printf("\n\nPhysical address(i.e,frame no &
                        offset):%d,%d",page[pno],off);
                }
                printf("\nDo you want to continue(1/0)?:");
                scanf("%d",&choice);
        }while(choice==1);
        return 1;
}
```

OUTPUT:

```
Enter the no of pages in memory: 2
Enter page size: 1
Enter no of frames: 2
Enter the page table
(Enter frame no as -1 if that page is not present in any frame)
pageno     frameno
-------     -------

0          2
1          1
Enter the logical address(i.e,page no & offset):0 2
Physical address(i.e,frame no & offset):2,2
Do you want to continue(1/0)?:0
```

RESULT:

Paging technique of memory management was simulated successfully.

EXPERIMENT-9
MEMORY ALLOCATION TECHNIQUES

FIRST-FIT MEMORY ALLOCATION TECHNIQUE:

AIM:

To stimulate the memory allocation technique of First Fit.

ALGORITHM:

Step 1: Start.
Step 2: Input the memory blocks and processes with their sizes respectively.
Step 3: Initialize all the memory blocks as free.
Step 4: Pick the process and find the first block with enough space that can be assigned to the current process.
Step 5: If not, then leave that process and go to the next step.
Step 6: Repeat the process 4 and 5 for all the processes.
Step 7: Display the process number, process size, block number, block size and fragment.
Step 8: Stop

CODE:

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

OUTPUT:

```
Memory Management Scheme - First Fit
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

File_no:        File_size :     Block_no:       Block_size:     Fragement
1               1               1               5               4
2               4               3               7               3
```

RESULT:

Memory allocation technique of First Fit was stimulated successfully.

BEST-FIT MEMORY ALLOCATION TECHNIQUE:

AIM:

To stimulate the memory allocation technique of Best Fit.

ALGORITHM:

Step 1: Start.
Step 2: Input the memory blocks and processes with their sizes respectively.
Step 3: Initialize all the memory blocks as free.
Step 4: Pick the process and find the minimum block size with enough space that can be assigned to the current process.
Step 5: If not, then leave that process and go to the next step.
Step 6: Repeat the process 4 and 5 for all the processes.
Step 7: Display the process number, process size, block number, block size and fragment.
Step 8: Stop

CODE:

```
#include<stdio.h>
int main()
{   int fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
    static int barray[20],parray[20];
    printf("\n\t\t\tMemory Management Scheme - Best Fit");
    printf("\nEnter the number of Memory blocks:");
    scanf("%d",&nb);
    printf("Enter the number of Files:");
    scanf("%d",&np);
    printf("\nEnter the size of the blocks:-\n");
    for(i=1;i<=nb;i++)
    {   printf("Block no.%d:",i);
        scanf("%d",&b[i]);
    }
    printf("\nEnter the size of the Files:-\n");
    for(i=1;i<=np;i++)
    {   printf("File no.%d:",i);
        scanf("%d",&p[i]);
    }
    for(i=1;i<=np;i++)
    {   for(j=1;j<=nb;j++)
        {   if(barray[j]!=1)
            {   temp=b[j]-p[i];
                if(temp>=0)
                if(lowest>temp)
                {   parray[i]=j;
                    lowest=temp;
                }
            }
        }
        fragment[i]=lowest;
        barray[parray[i]]=1;
        lowest=10000;
    }
    printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
    for(i=1;i<=np && parray[i]!=0;i++)
    printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
    ("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

OUTPUT:

```
Memory Management Scheme - Best Fit
Enter the number of blocks:3
Enter the number of processes:2
Enter the size of the blocks:-
Block no.1:5
Block no.2:2
Block no.3:7
Enter the size of the processes :-
Process no.1:1
Process no.2:4

Process_no  Process_size   Block_no  Block_size   Fragment
1                1              2          2             1
2                4              1          5             1
```

RESULT:

Memory allocation technique of Best Fit was stimulated successfully.

WORST-FIT MEMORY ALLOCATION TECHNIQUE

AIM:

To stimulate the memory allocation technique of Worst Fit.

ALGORITHM:

Step 1: Start.
Step 2: Input the memory blocks and processes with their sizes respectively.
Step 3: Initialize all the memory blocks as free.
Step 4: Pick the process and find the maximum block size with enough space that can be assigned to the current process.
Step 5: If not, then leave that process.
Step 6: Repeat the process 4 and 5 for all the processes.
Step 7: Display the process number, process size, block number, block size and fragment.
Step 8: Stop.

CODE:

```
#include<stdio.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];
    printf("\n\tMemory Management Scheme - Worst Fit");
    printf("\nEnter the number of Memory 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(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]);
}
```

OUTPUT:

```
Memory Management Scheme - Worst Fit
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

File_no:        File_size :     Block_no:       Block_size:     Fragement
1               1               3               7               6
2               4               1               5               1
```

RESULT:

Memory allocation technique of Worst Fit was stimulated successfully.

SINGLE LEVEL DIRECTORY

AIM:

To simulate a program for Single Level Directory.

ALGORITHM:

```
Step 1:Start
Step 2: Get all the necessary input from the keyboard.
Step 3: Enter 1 to Create a file.
Step 3.1: Enter 2 to Delete the specified file.
Step 3.2: Enter 3 to Search for the specified file.
Step 3.3: Enter 4 to Display the name of the files present.
Step 3.4: Enter 5 to exit.
Step 4: repeat step 3 until you complete the operations in the directory.
Step 5: Stop.
```

CODE:

```
#include<stdio.h>
#include<string.h>
struct
{   char dname[10],fname[10][10];
    int fcnt;
}dir;
void main()
{   int i,ch;
    char f[30];
    dir.fcnt = 0;
    printf("\nEnter name of directory -- ");
    scanf("%s", dir.dname);
    while(1)
    {   printf("\n\n1. Create File\t2. Delete File\t3. Search File \n4. Display Files\t5.
Exit\nEnter your choice -- ");
        scanf("%d",&ch);
        switch(ch)
        {   case 1:
                printf("\nEnter the name of the file -- ");
                scanf("%s",dir.fname[dir.fcnt]);
                dir.fcnt++;
                break;
            case 2:
                printf("\nEnter 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("\nEnter 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("\nDirectory Empty");
                else
                {   printf("\nThe Files are -- ");
                    for(i=0;i<dir.fcnt;i++)
                    printf("\t%s",dir.fname[i]);
                }
                break;
            default: exit(0);
        }
    }
}
```

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 File3. 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 File3. 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 File3. Search File
4. Display Files        5. Exit
Enter your choice -- 5
```

RESULT:

Program for the Single Level Directory was simulated successfully.

TWO LEVEL DIRECTORY:

AIM:

To simulate a program for Two Level Directory.

ALGORITHM:

```
Step 1: Start
Step 2: Get all the necessary input from the keyboard.
Step 3: Enter 1 to Create a Directory.
Step 3.1: Enter 2 to Create a file inside the specified Directory.
Step 3.2: Enter 3 to delete the specified file in the specified directory.
Step 3.3: Enter 4 to Search for a file in the specified directory.
Step 3.4: Enter 5 to Display the no of directories and no of files in them.
Step 3.5: Enter 6 to Exit.
Step 4: Repeat step 3 until you complete the desired operations in the directory.
Step 5: Stop.
```

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];
    dcnt=0;
    while(1)
    {   printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");
        printf("\n4. Search File\t\t5. Display\t6. Exit\t Enter your choice --");
        scanf("%d",&ch);
        switch(ch)
        {   case 1:
                printf("\nEnter name of directory -- ");
                scanf("%s", dir[dcnt].dname);
                dir[dcnt].fcnt=0;
                dcnt++;
                printf("Directory created");
                break;
            case 2:
                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",dir[i].fname[dir[i].fcnt]);
                    dir[i].fcnt++;
                    printf("File created");
                }
                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++)
                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);
        }}}
```

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 -- AI
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 -- CSE
File created

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 -- AI
File AI is deleted

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

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

RESULT:

Program for the Two-Level Directory was simulated successfully.

EXPERIMENT-10
PAGE REPLACEMENT ALGORITHMS

FIRST IN FIRST OUT ALGORITHM:

AIM:

To Simulate FIFO Algorithm in Paging Technique of memory management.

ALGORITHM:

Step 1: Start.
Step 2: Start to traverse the pages.
Step 3: If the memory holds fewer pages, then the capacity else goes to step 5.
Step 4: Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.
Step 5: If the current page is present in the memory, do nothing.
Step 6: Else, pop the topmost page from the queue as it was inserted first.
Step 7: Replace the topmost page with the current page from the string.
Step 8: Increment the page faults.
Step 9: Stop.

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:");
    scanf("%d",&n);
    printf("\n ENTER THE REFERENCE STRING:");
    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]= 0;
          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 Faults= %d",count);
           return 0;
}
```

OUTPUT:

```
ENTER THE NUMBER OF PAGES:12
ENTER THE REFERENCE STRING:1 2 3 4 5 1 2 5 1 2 3 4
ENTER THE NUMBER OF FRAMES :3
ref string        page frames
1                 1        -1        -1
2                 1        2         -1
3                 1        2         3
4                 4        2         3
5                 4        5         3
1                 4        5         1
2                 2        5         1
5
1
2
3                 2        3         1
4                 2        3         4
Page Fault Is 9
```

RESULT:

First In First Out algorithm of paging technique in memory management was simulated successfully.

OPTIMAL PAGE REPLACEMENT ALGORITHM

AIM:

To Simulate Optimal Page Replacement Algorithm in Paging Technique of memory management.

ALGORITHM:

Step 1: Start.
Step 2: Push the first page in the stack as per the memory demand.
Step 3: Push the second page as per the memory demand.
Step 4: Push the third page until the memory is full.
Step 5: As the queue is full, the page which is least recently used is popped.
Step 6: repeat step 4 until the page demand continues and until the processing is over.
Step 7: Terminate the program.
Step 8: Stop.

CODE:

```
#include<stdio.h>
int main()
{   int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i,
j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);
    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);
    printf("Enter page reference string: ");
    for(i = 0; i < no_of_pages; ++i)
    {   scanf("%d", &pages[i]);
    }
    for(i = 0; i < no_of_frames; ++i)
    {   frames[i] = -1;
    }
    for(i = 0; i < no_of_pages; ++i)
    {   flag1 = flag2 = 0;
        for(j = 0; j < no_of_frames; ++j)
        {   if(frames[j] == pages[i])
            {   flag1 = flag2 = 1;
                break;
            }
        }
        if(flag1 == 0)
        {   for(j = 0; j < no_of_frames; ++j)
            {   if(frames[j] == -1)
                {   faults++;
                    frames[j] = pages[i];
                    flag2 = 1;
                    break;
                }
            }
        }
        if(flag2 == 0)
        {   flag3 =0;
            for(j = 0; j < no_of_frames; ++j)
            {   temp[j] = -1;
                for(k = i + 1; k < no_of_pages; ++k)
                {   if(frames[j] == pages[k])
                    {   temp[j] = k;
                        break;
                    }
                }
            }
            for(j = 0; j < no_of_frames; ++j)
            {   if(temp[j] == -1)
                {   pos = j;
                    flag3 = 1;
                    break;
                }
            }
            if(flag3 ==0)
            {   max = temp[0];
                pos = 0;
                for(j = 1; j < no_of_frames; ++j)
                {   if(temp[j] > max)
                    {   max = temp[j];
                        pos = j;
                    }
                }
            }
            frames[pos] = pages[i];
            faults++;
        }
        printf("\n");
        for(j = 0; j < no_of_frames; ++j)
        {   printf("%d\t", frames[j]);
        }
    }
    printf("\n\nTotal Page Faults = %d", faults);
    return 0;
}
```

OUTPUT:

```
Enter number of frames: 3
Enter number of pages: 12
Enter page reference string: 1 2 3 4 5 1 2 5 1 2 3 4
1       -1      -1
```

```
1       2       -1
1       2       3
1       2       4
1       2       5
1       2       5
1       2       5
1       2       5
1       2       5
1       2       5
3       2       5
4       2       5

Total Page Faults = 7
```

RESULT:

Optimal Page Replacement algorithm of paging technique in memory management was simulated successfully.