OPERATING SYSTEMS

LAB MANUAL

Subject Code:	A30517
Regulations:	R18
Class:	III Year I Semester (CSE)

Prepared By
Mrs Swati M Ollalwar
Associate Professor

Department of Computer Science & Engineering

CMR COLLEGE OF ENGINEERING & TECHNOLOGY

Medchal – 501 401, Hyderabad

OPERATING SYSTEMS LAB SYLLABUS

Implement the following programs using C language.

OPERATING SYSTEMS LABORATORY

OBJECTIVE:

This lab complements the operating systems course. Students will gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, memory management, file systems and deadlock handling using C language in Linux environment.

OUTCOMES:

Upon the completion of Operating Systems practical course, the student will be able to:

1. **Understand** and implement basic services and functionalities of the operating system using system calls.
2. **Use** modern operating system calls and synchronization libraries in software/ hardware interfaces**.**
3. **Understand** the benefits of thread over process and implement synchronized programs using multithreading concepts.
4. **Analyze** and simulate CPU Scheduling Algorithms like FCFS, Round Robin, SJF, and Priority.
5. **Implement** memory management schemes and page replacement schemes.
6. **Simulate** file allocation and organization techniques.
7. **Understand** the concepts of deadlock in operating systems and implement them in multiprogramming system.

EXPERIMENT 1

1.1 OBJECTIVE

Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time for the above problem.
a) FCFS b) SJF c) Round Robin d) Priority

1.2 DESCRIPTION

Assume all the processes arrive at the same time.

1.2.1 FCFS CPU SCHEDULING ALGORITHM

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

1.2.2 SJF CPU SCHEDULING ALGORITHM

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

1.2.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

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

1.2.4 PRIORITY CPU SCHEDULING ALGORITHM

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

1.3 PROGRAM

1.3.1 FCFS CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
        int bt[20], wt[20], tat[20], i, n;
        float wtavg, tatavg;
        clrscr();
        printf("\nEnter the number of processes -- ");
                scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter Burst Time for Process %d -- ", i);
                scanf("%d", &bt[i]);
        }
        wt[0] =  wtavg = 0;
        tat[0] = tatavg = bt[0];
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] +bt[i-1];
                tat[i] = tat[i-1] +bt[i];
                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }
        printf("\t PROCESS \tBURST TIME \t WAITING TIME\t
                                    TURNAROUND TIME\n");

        for(i=0;i<n;i++)
                printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
        printf("\nAverage Waiting Time -- %f", wtavg/n); printf("\nAverage
        Turnaround Time -- %f", tatavg/n); getch();

}
```

INPUT

```
Enter the number of processes --    3
Enter Burst Time for Process 0 --   24
Enter Burst Time for Process 1 --   3
Enter Burst Time for Process 2 --   3
```

OUTPUT

```
PROCESS         BURST TIME          WAITING TIME        TURNAROUND TIME
P0                 24                   0                    24
P1                  3                  24                    27
P2                  3                  27                    30

Average Waiting Time--   17.000000
Average Turnaround Time --          27.000000
```

1.3.2 SJF CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
        int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
        float wtavg, tatavg;
        clrscr();
        printf("\nEnter the number of processes -- ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                p[i]=i;
                printf("Enter Burst Time for Process %d -- ", i);
                scanf("%d", &bt[i]);

        }
        for(i=0;i<n;i++)
                for(k=i+1;k<n;k++)
                        if(bt[i]>bt[k])
                        {
                                temp=bt[i];
                                bt[i]=bt[k];
                                bt[k]=temp;

                                temp=p[i];
                                p[i]=p[k];
                                p[k]=temp;
                        }
        wt[0] =  wtavg = 0;
        tat[0] = tatavg = bt[0];
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] +bt[i-1];
                tat[i] = tat[i-1] +bt[i];
                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }
        printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
        for(i=0;i<n;i++)
                printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i],
        tat[i]); printf("\nAverage Waiting Time -- %f", wtavg/n);
        printf("\nAverage Turnaround Time -- %f", tatavg/n);
        getch();
}
```

INPUT

```
Enter the number of processes --    4
Enter Burst Time for Process 0 --   6
Enter Burst Time for Process 1 --   8
Enter Burst Time for Process 2 --   7
Enter Burst Time for Process 3 --   3
```

OUTPUT

```
PROCESS            BURST TIME         WAITING TIME    TURNAROUND TIME
P3                     3                     0               3
P0                     6                     3               9
P2                     7                     9               16
P1                     8                     16              24
Average Waiting Time --          7.000000
Average Turnaround Time --       13.000000
```

1.3.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
        float awt=0,att=0,temp=0;
        clrscr();
        printf("Enter the no of processes -- ");
        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
                printf("\nEnter Burst Time for process %d -- ", i+1);
                scanf("%d",&bu[i]);
                ct[i]=bu[i];
        }
        printf("\nEnter the size of time slice -- ");
        scanf("%d",&t);
        max=bu[0];
        for(i=1;i<n;i++)
                if(max<bu[i])
                        max=bu[i];
        for(j=0;j<(max/t)+1;j++)
                for(i=0;i<n;i++)
                        if(bu[i]!=0)
                                if(bu[i]<=t)
                                {
                                        tat[i]=temp+bu[i];
                                        temp=temp+bu[i];
                                        bu[i]=0;
                                }
```

```
                                                  else
                                                  {
                                                            bu[i]=bu[i]-t;
                                                            temp=temp+t;
                                                   }
            for(i=0;i<n;i++)
            {
                        wa[i]=tat[i]-ct[i];
                        att+=tat[i];
                        awt+=wa[i];
            }
            printf("\nThe Average Turnaround time is -- %f",att/n);
            printf("\nThe Average Waiting time is -- %f ",awt/n);
            printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
            for(i=0;i<n;i++)
                        printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
            getch();
}
```

INPUT

```
Enter the no of processes – 3
Enter Burst Time for process 1 –     24
Enter Burst Time for process 2 --    3
Enter Burst Time for process 3 --    3

Enter the size of time slice – 3
```

OUTPUT

```
The Average Turnaround time is – 15.666667
The Average Waiting time is --       5.666667

PROCESS            BURST TIME           WAITING TIME            TURNAROUND TIME
1                     24                      6                              30
2                     3                       4                              7
3                     3                       7                              10
```

1.3.4 ***PRIORITY CPU SCHEDULING ALGORITHM***

```
#include<stdio.h>
main()
{
          int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
          float wtavg, tatavg;
          clrscr();
          printf("Enter the number of processes --- ");
          scanf("%d",&n);

          for(i=0;i<n;i++)
          {
                    p[i] = i;
                    printf("Enter the Burst Time & Priority of Process %d --- ",i);
                    scanf("%d %d",&bt[i], &pri[i]);
          }
          for(i=0;i<n;i++)
                    for(k=i+1;k<n;k++)
                              if(pri[i] > pri[k])
                              {
                                        temp=p[i];
                                        p[i]=p[k];
                                        p[k]=temp;
```

```
                                    temp=bt[i];
                                    bt[i]=bt[k];
                                    bt[k]=temp;

                                    temp=pri[i];
                                    pri[i]=pri[k];
                                    pri[k]=temp;

                              }
            wtavg = wt[0] = 0;
            tatavg = tat[0] = bt[0]; for(i=1;i<n;i++)
            {
                  wt[i] = wt[i-1] + bt[i-1];
                  tat[i] = tat[i-1] + bt[i];

                  wtavg = wtavg + wt[i];
                  tatavg = tatavg + tat[i];
            }

            printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
            for(i=0;i<n;i++)
                  printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);

            printf("\nAverage Waiting Time is --- %f",wtavg/n);
            printf("\nAverage Turnaround Time is --- %f",tatavg/n);
            getch();
}
```

INPUT

```
Enter the number of processes --  5
Enter the Burst Time & Priority of Process 0 --- 10     3
Enter the Burst Time & Priority of Process 1 --- 1      1
Enter the Burst Time & Priority of Process 2 --- 2      4
Enter the Burst Time & Priority of Process 3 --- 1      5
Enter the Burst Time & Priority of Process 4 --- 5      2
```

OUTPUT

```
PROCESS        PRIORITY       BURST TIME          WAITING TIME    TURNAROUND TIME
1              1                 1                    0                1
4              2                 5                    1                6
0              3                 10                   6                16
2              4                 2                    16               18
3              5                 1                    18               19
Average Waiting Time is ---   8.200000
Average Turnaround Time is  --- 12.000000
```

EXPERIMENT 2

2.1 OBJECTIVE

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

2.2 DESCRIPTION

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

2.3 PROGRAM

```
#include<stdio.h>
struct file
{
        int all[10];
        int max[10];
        int need[10];
        int flag;
};
void main()
{
        struct file f[10];
        int fl;
        int i, j, k, p, b, n, r, g, cnt=0, id, newr;
        int avail[10],seq[10];
        clrscr();
        printf("Enter number of processes -- ");
        scanf("%d",&n);
        printf("Enter number of resources -- ");
        scanf("%d",&r);
        for(i=0;i<n;i++)
        {
                printf("Enter details for P%d",i);
                printf("\nEnter allocation\t -- \t");
                for(j=0;j<r;j++)
                        scanf("%d",&f[i].all[j]);
                printf("Enter Max\t\t -- \t");
                for(j=0;j<r;j++)
                        scanf("%d",&f[i].max[j]);
                f[i].flag=0;
        }
        printf("\nEnter Available Resources\t -- \t");
        for(i=0;i<r;i++)
                scanf("%d",&avail[i]);

        printf("\nEnter New Request Details -- ");
        printf("\nEnter pid \t -- \t");
        scanf("%d",&id);
        printf("Enter Request for Resources \t -- \t");
        for(i=0;i<r;i++)
        {
                scanf("%d",&newr);
                f[id].all[i] += newr;
                avail[i]=avail[i] - newr;
        }

        for(i=0;i<n;i++)
        {
                for(j=0;j<r;j++)
                {
                        f[i].need[j]=f[i].max[j]-f[i].all[j];
                        if(f[i].need[j]<0)
                                f[i].need[j]=0;
                }
        }
        cnt=0;
        fl=0;
        while(cnt!=n)
        {
                g=0;
                for(j=0;j<n;j++)
                {
                        if(f[j].flag==0)
                        {
                                b=0;
                                for(p=0;p<r;p++)
                                {
                                        if(avail[p]>=f[j].need[p])
                                                b=b+1;
                                        else
                                                b=b-1;
                                }
                                if(b==r)
                                {
                                        printf("\nP%d is visited",j);
                                        seq[fl++]=j;
                                        f[j].flag=1;
                                        for(k=0;k<r;k++)
                                                avail[k]=avail[k]+f[j].all[k];
                                        cnt=cnt+1;
                                        printf("(");
                                        for(k=0;k<r;k++)
                                                printf("%3d",avail[k]);
                                        printf(")");
                                        g=1;
                                }
                        }
                }
                if(g==0)
                {
                        printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
                        printf("\n SYSTEM IS IN UNSAFE STATE"); goto y;

                }
        }
        printf("\nSYSTEM IS IN SAFE STATE");
        printf("\nThe Safe Sequence is -- (");
        for(i=0;i<fl;i++)
                printf("P%d ",seq[i]);
        printf(")");
      y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
        for(i=0;i<n;i++)
        {
                printf("P%d\t",i);
                for(j=0;j<r;j++)
```

```
                    printf("%6d",f[i].all[j]);
            for(j=0;j<r;j++)
                    printf("%6d",f[i].max[j]);
            for(j=0;j<r;j++)
                    printf("%6d",f[i].need[j]);
            printf("\n");
        }
        getch();
}
```

INPUT

```
Enter number of processes        –        5
Enter number of resources        --       3
Enter details for P0
Enter allocation              --     0       1       0
Enter Max                     --             7       5       3

Enter details for P1
Enter allocation              --     2       0       0
Enter Max                     --     3       2       2

Enter details for P2
Enter allocation              --     3       0       2
Enter Max                     --     9       0       2

Enter details for P3
Enter allocation              --     2       1       1
Enter Max                     --     2       2       2

Enter details for P4
Enter allocation              --     0       0       2
Enter Max                     --     4       3       3

Enter Available Resources --    3  3       2
Enter New Request Details --
Enter pid      --    1
Enter Request for Resources     --  1       0      2
```

OUTPUT

```
P1 is visited( 5  3   2)
P3 is visited( 7  4   3)
P4 is visited( 7  4   5)
P0 is visited( 7  5   5)
P2 is visited( 10    5  7)
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- (P1 P3 P4 P0 P2 )

Process         Allocation                Max                  Need
P0              0    1    0            7    5    3          7    4 3
P1              3    0    2            3    2    2          0    2 0
P2              3    0    2            9    0    2          6    0 0
P3              2    1    1            2    2    2          0    1 1
P4              0    0    2            4    3    3          4    3 1
```

EXPERIMENT 3

3.1 OBJECTIVE

a) Write a C program to simulate paging technique of memory management.
b) Write a C program to simulate segmentation technique of memory management.

3.2 DESCRIPTION

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

PAGING TECHNIQUE

3.3.1 PROGRAM

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

main()
{
        int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
        int s[10], fno[10][20];

        clrscr();

        printf("\nEnter the memory size -- ");
        scanf("%d",&ms);

        printf("\nEnter the page size -- ");
        scanf("%d",&ps);

        nop = ms/ps;
        printf("\nThe no. of pages available in memory are -- %d ",nop);

        printf("\nEnter number of processes -- ");
        scanf("%d",&np);

        rempages = nop;

        for(i=1;i<=np;i++)
        {

                printf("\nEnter no. of pages required for p[%d]-- ",i);
                scanf("%d",&s[i]);

                if(s[i] >rempages)
                {
                        printf("\nMemory is Full");
                        break;
                }
                rempages = rempages - s[i];

                printf("\nEnter pagetable for p[%d] --- ",i);
                for(j=0;j<s[i];j++)
                        scanf("%d",&fno[i][j]);
        }
```

```
		printf("\nEnter Logical Address to find Physical Address ");
		printf("\nEnter process no. and pagenumber and offset -- ");

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

INPUT

```
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
```

OUTPUT

```
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
```

SEGMENTATION TECHNIQUE

3.3.2 PROGRAM

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int sost;
void gstinfo();
void ptladdr();

struct segtab
{
	int sno;
	int baddr;
	int limit;
	int val[10];
}st[10];

void gstinfo()
{
	int i,j;
	printf("\n\tEnter the size of the segment table: ");
	scanf("%d",&sost);

	for(i=1;i<=sost;i++)
		{
		printf("\n\tEnter the information about segment: %d",i);
		st[i].sno = i;
```

```
                printf("\n\tEnter the base Address: ");
                scanf("%d",&st[i].baddr);
                printf("\n\tEnter the Limit: ");
                scanf("%d",&st[i].limit);
                for(j=0;j<st[i].limit;j++)
                        {
                        printf("Enter the %d address Value: ",(st[i].baddr + j));
                        scanf("%d",&st[i].val[j]);
                        }
                }
}

void ptladdr()
{
        int i,swd,d=0,n,s,disp,paddr;
        clrscr();
        printf("\n\n\t\t\t SEGMENT TABLE \n\n");
        printf("\n\t      SEG.NO\tBASE ADDRESS\t      LIMIT \n\n");
        for(i=1;i<=sost;i++)
                printf("\t\t%d    \t\t%d\t\t%d\n\n",st[i].sno,st[i].baddr,st[i].limit);
        printf("\n\nEnter the logical Address: ");
        scanf("%d",&swd);
        n=swd;
        while (n != 0)
        {
                n=n/10;
                d++;
        }

        s = swd/pow(10,d-1);
        disp = swd%(int)pow(10,d-1);

        if(s<=sost)
        {
                if(disp < st[s].limit)
                {
                        paddr = st[s].baddr + disp;
                        printf("\n\t\tLogical Address is: %d",swd);
                        printf("\n\t\tMapped Physical address is: %d",paddr);
                        printf("\n\tThe value is: %d",( st[s].val[disp] )  );
                }
                else
                        printf("\n\t\tLimit of segment %d is high\n\n",s);
        }

        else
                printf("\n\t\tInvalid Segment Address \n");
        }

        void main()
        {
        char ch;
        clrscr();
        gstinfo();
        do
        {
        ptladdr();
        printf("\n\t Do U want to Continue(Y/N)");
        flushall();
        scanf("%c",&ch);
        }while (ch == 'Y' || ch == 'y' );

        getch();
        }
```

OUTPUT:

```
        Enter the size of the segment table: 3
        Enter the information about segment: 1
        Enter the base Address: 4
        Enter the Limit: 5
Enter the 4 address Value: 11
Enter the 5 address Value: 12
Enter the 6 address Value: 13
Enter the 7 address Value: 14
Enter the 8 address Value: 15

        Enter the information about segment: 2
        Enter the base Address: 5
        Enter the Limit: 4
Enter the 5 address Value: 21
Enter the 6 address Value: 31
Enter the 7 address Value: 41
Enter the 8 address Value: 51
        Enter the information about segment: 3
        Enter the base Address: 3
        Enter the Limit: 4
Enter the 3 address Value: 31
Enter the 4 address Value: 41
Enter the 5 address Value: 41
Enter the 6 address Value: 51

SEGMENT TABLE

              SEG.NO    BASE ADDRESS         LIMIT

                1                4                  5

                2                 5                4

                3                3                 4

                 Enter the logical Address: 3
                Logical Address is: 3
                Mapped Physical address is: 3
              The value is: 31
              Do U want to Continue(Y/N)


                         SEGMENT TABLE


              SEG.NO    BASE ADDRESS         LIMIT

                1                 4                5

                2                 5                4

                3                3                 4

              Enter the logical Address: 1

                Logical Address is: 1
                Mapped Physical address is: 4
               The value is: 11
               Do U want to Continue(Y/N)
```

EXPERIMENT 4

4.1 OBJECTIVE

Write a C program to simulate page replacement algorithms
a) FIFO b) LRU c) OPTIMAL

4.2 DESCRIPTION

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. The basic idea is to replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

4.3 PROGRAM

4.3.1 FIFO PAGE REPLACEMENT ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
        clrscr();
        printf("\n Enter the length of reference string -- ");
        scanf("%d",&n);
        printf("\n Enter the reference string -- ");
        for(i=0;i<n;i++)
                scanf("%d",&rs[i]);
        printf("\n Enter no. of frames -- ");
        scanf("%d",&f);
        for(i=0;i<f;i++)
                m[i]=-1;

        printf("\n The Page Replacement Process is -- \n");
        for(i=0;i<n;i++)
        {
                for(k=0;k<f;k++)
                {
                        if(m[k]==rs[i])
                                break;
                }
                if(k==f)
                {
                        m[count++]=rs[i];
                        pf++;
                }
                for(j=0;j<f;j++)
                        printf("\t%d",m[j]);
                if(k==f)
                        printf("\tPF No. %d",pf);
                printf("\n");
                if(count==f)
                        count=0;
        }

        printf("\n The number of Page Faults using FIFO are %d",pf);
        getch();
}
```

INPUT

```
Enter the length of reference string – 20
Enter the reference string --                   70120304230321201701
Enter no. of frames --      3
```

OUTPUT

```
The Page Replacement Process is –
     7   -1    -1            PF No. 1
     7    0    -1            PF No. 2
     7    0     1            PF No. 3
     2    0     1            PF No. 4
     2    0     1
     2    3     1            PF No. 5
     2    3     0            PF No. 6
     4    3     0            PF No. 7
     4    2     0            PF No. 8
     4    2     3            PF No. 9
     0    2     3            PF No. 10
     0    2     3
     0    2     3
     0    1     3            PF No. 11
     0    1     2            PF No. 12
     0    1     2
     0    1     2
     7    1     2            PF No. 13
     7    0     2            PF No. 14
     7    0     1            PF No. 15

The number of Page Faults using FIFO are 15
```

4.3.2 LRU PAGE REPLACEMENT ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1; clrscr();
        printf("Enter the length of reference string -- ");
        scanf("%d",&n);
        printf("Enter the reference string -- ");
        for(i=0;i<n;i++)
        {
                scanf("%d",&rs[i]);
                flag[i]=0;
        }
        printf("Enter the number of frames -- ");
        scanf("%d",&f);
        for(i=0;i<f;i++)
        {
                count[i]=0;
                m[i]=-1;
        }
        printf("\nThe Page Replacement process is -- \n");
```

```
            for(i=0;i<n;i++)
            {
                        for(j=0;j<f;j++)
                        {


                                    if(m[j]==rs[i])
                                    {
                                                flag[i]=1;
                                                count[j]=next;
                                                next++;
                                    }

                        }
                        if(flag[i]==0)
                        {
                                    if(i<f)
                                    {
                                                m[i]=rs[i];
                                                count[i]=next;
                                                next++;
                                    }
                                    else
                                    {
                                                min=0;
                                                for(j=1;j<f;j++)
                                                                      if(count[min] > count[j])
                                                                    min=j;

                                                m[min]=rs[i];
                                                count[min]=next;
                                                next++;
                                    }
                                    pf++;
                        }
                        for(j=0;j<f;j++)
                                    printf("%d\t", m[j]);
                        if(flag[i]==0)
                                    printf("PF No. -- %d" , pf);
                        printf("\n");
            }
            printf("\nThe number of page faults using LRU are %d",pf); getch();
}
```

INPUT

```
Enter the length of reference string -- 20
Enter the reference string -- 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
```

OUTPUT

```
The Page Replacement process is --
  7    -1     -1     PF No. -- 1
  7    0      -1     PF No. -- 2
  7    0      1      PF No. -- 3
  2    0      1      PF No. -- 4
  2    0      1
  2    0      3      PF No. -- 5
  2    0      3
  4    0      3      PF No. -- 6
  4    0      2      PF No. -- 7
  4    3      2      PF No. -- 8
  0    3      2      PF No. -- 9
  0    3      2
  0    3      2
  1    3      2      PF No. -- 10
  1    3      2
  1    0      2      PF No. -- 11
  1    0      2
  1    0      7      PF No. -- 12
  1    0      7
  1    0      7
The number of page faults using LRU are 12
```

4.3.3. OPTIMAL PAGE REPLACEMENT ALGORITHM

```
#include<stdio.h>
int n;
main()
{
        int seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s;
        int count=1,pf=0,p=0;
        float pfr;
        clrscr();
        printf("Enter maximum limit of the sequence: ");
        scanf("%d",&max);
        printf("\nEnter the sequence: ");
        for(i=0;i<max;i++)
                scanf("%d",&seq[i]);
                printf("\nEnter no. of frames: ");
                scanf("%d",&n);
                fr[0]=seq[0];
                pf++;
                printf("%d\t",fr[0]);
                i=1;
                while(count<n)
                {
                        flag=1;
                        p++;
                        for(j=0;j<i;j++)
                        {
                                if(seq[i]==seq[j])
                                flag=0;
                        }
                        if(flag!=0)
                        {
                                fr[count]=seq[i];
                                printf("%d\t",fr[count]);
                                count++;
                                pf++;
                        }
```

```
                    i++;
          }
          printf("\n");



          for(i=p;i<max;i++)
          {
                    flag=1;
                    for(j=0;j<n;j++)
                    {
                              if(seq[i]==fr[j])
                                        flag=0;
                    }
                    if(flag!=0)
                    {
                              for(j=0;j<n;j++)
                              {
                                        m=fr[j];
                                        for(k=i;k<max;k++)
                                        {
                                                  if(seq[k]==m)
                                                  {
                                                            pos[j]=k;
                                                            break;
                                                  }
                                                  else
                                                                        pos[j]=1;
                                              }
                              }
                              for(k=0;k<n;k++)
                              {
                                        if(pos[k]==1)
                                                  flag=0;
                              }
                              if(flag!=0)
                                        s=findmax(pos);
                              if(flag==0)
                              {
                                        for(k=0;k<n;k++)
                                        {
                                                  if(pos[k]==1)
                                                  {
                                                            s=k;
                                                            break;
                                                  }
                                              }
                              }
                              fr[s]=seq[i];
                              for(k=0;k<n;k++)
                                        printf("%d\t",fr[k]);
                              pf++;
                              printf("\n");
                    }
          }
          pfr=(float)pf/(float)max;
          printf("\nThe no. of page faults are %d",pf);
          printf("\nPage fault rate %f",pfr);
          getch();
}
```

```
int findmax(int a[])
{
        int max,i,k=0;
        max=a[0];
        for(i=0;i<n;i++)
        {
                if(max<a[i])
                {
                        max=a[i];
                        k=i;
                }
        }
        return k;
}
```

INPUT

```
Enter number of page references -- 10
Enter the reference string --                    123452525143

Enter the available no. of frames -- 3
```

OUTPUT

```
The Page Replacement Process is –

1       -1      -1              PF No. 1
1       2       -1              PF No. 2
1       2       3               PF No. 3
4       2       3               PF No. 4
5       2       3               PF No. 5
5       2       3
5       2       3
5       2       1               PF No. 6
5       2       4               PF No. 7
5       2       3               PF No. 8

Total number of page faults --  8
```

EXPERIMENT 5

5.1 OBJECTIVE

Write a C program to simulate the following file allocation strategies.
a) Sequential b) Linked c)) Indexed

5.2 DESCRIPTION

A file is a collection of data, usually stored on disk. As a logical entity, a file enables to divide data into meaningful groups. As a physical entity, a file should be considered in terms of its organization. The term "file organization" refers to the way in which data is stored in a file and, consequently, the method(s) by which it can be accessed.

5.2.1 SEQUENTIAL FILE ALLOCATION

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record. A record of a sequential file can only be accessed by reading all the previous records.

5.2.2 LINKED FILE ALLOCATION

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.

5.2.3 INDEXED FILE ALLOCATION

Indexed file allocation strategy brings all the pointers together into one location: an index block. Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block. To find and read the i^{th} block, the pointer in the i^{th} index-block entry is used.

5.3 PROGRAM

5.3.1 SEQUENTIAL FILE ALLOCATION

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

struct fileTable
{
        char name[20];
        int sb, nob;
}ft[30];

void main()
{
        int i, j, n;
        char s[20];
        clrscr();
        printf("Enter no of files   :");
        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
                printf("\nEnter file name %d           :",i+1);
                scanf("%s",ft[i].name);
                printf("Enter starting block of file %d           :",i+1);
                scanf("%d",&ft[i].sb);
                printf("Enter no of blocks in file %d :",i+1);
                scanf("%d",&ft[i].nob);
        }
        printf("\nEnter the file name to be searched -- ");
        scanf("%s",s);
        for(i=0;i<n;i++)
                if(strcmp(s, ft[i].name)==0)
```

```
                    break;
        if(i==n)
                printf("\nFile Not Found");
        else
        {
                printf("\nFILE NAME START BLOCK NO OF BLOCKS BLOCKS OCCUPIED\n");
                printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob);
                for(j=0;j<ft[i].nob;j++)
                        printf("%d, ",ft[i].sb+j);
        }
        getch();
}
```

INPUT:

```
Enter no of files  :3

Enter file name 1   :A
Enter starting block of file 1 :85
Enter no of blocks in file 1   :6

Enter file name 2   :B
Enter starting block of file 2 :102
Enter no of blocks in file 2   :4

Enter file name 3   :C
Enter starting block of file 3 :60
Enter no of blocks in file 3   :4
Enter the file name to be searched -- B
```

OUTPUT:

```
FILE NAME         START BLOCK        NO OF BLOCKS        BLOCKS OCCUPIED
B                    102                  4              102, 103, 104, 105
```

5.3.2 LINKED FILE ALLOCATION

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

struct fileTable
{
char name[20];
int nob;
struct block *sb;
}ft[30];

struct block
{
        int bno;
        struct block *next;
};

void main()
{
        int i, j, n;
        char s[20];
        struct block *temp;
        clrscr();
        printf("Enter no of files      :");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter file name %d           :",i+1);
                scanf("%s",ft[i].name);
                printf("Enter no of blocks in file %d :",i+1);
                scanf("%d",&ft[i].nob);
                ft[i].sb=(struct block*)malloc(sizeof(struct block));
                temp = ft[i].sb;
                printf("Enter the blocks of the file   :");
                scanf("%d",&temp->bno);
                temp->next=NULL;

                for(j=1;j<ft[i].nob;j++)
                {
                        temp->next = (struct block*)malloc(sizeof(struct block));
                        temp = temp->next;
                        scanf("%d",&temp->bno);
                }
                temp->next = NULL;
        }
        printf("\nEnter the file name to be searched --  ");
        scanf("%s",s);
        for(i=0;i<n;i++)
                if(strcmp(s, ft[i].name)==0)
                        break;
        if(i==n)
                printf("\nFile Not Found");
        else
        {
                printf("\nFILE NAME NO OF BLOCKS  BLOCKS OCCUPIED");
                printf("\n   %s\t\t%d\t",ft[i].name,ft[i].nob);
                temp=ft[i].sb;
                for(j=0;j<ft[i].nob;j++)
                {
                        printf("%d → ",temp->bno);
                        temp = temp->next;
                }
        }
        getch();
}
```

INPUT:

```
Enter no of files   : 2

Enter file 1    : A
Enter no of blocks in file 1      : 4
Enter the blocks of the file 1          : 122394

Enter file 2    : G
Enter no of blocks in file 2      : 5
Enter the blocks of the file 2          :8877665544

Enter the file to be searched  : G
```

OUTPUT:

```
FILE NAME        NO OF BLOCKS                    BLOCKS OCCUPIED

G                          5                  88 → 77 → 66 → 55 → 44
```

5.3.3 *INDEXED FILE ALLOCATION*

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

struct fileTable
{
char name[20];
int nob, blocks[30];
}ft[30];

void main()
{
        int i, j, n;
        char s[20];
        clrscr();
        printf("Enter no of files     :");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter file name %d          :",i+1);
                scanf("%s",ft[i].name);
                printf("Enter no of blocks in file %d :",i+1);
                scanf("%d",&ft[i].nob);
                printf("Enter the blocks of the file   :");
                for(j=0;j<ft[i].nob;j++)
                        scanf("%d",&ft[i].blocks[j]);
        }

        printf("\nEnter the file name to be searched -- ");
        scanf("%s",s);
        for(i=0;i<n;i++)
                if(strcmp(s, ft[i].name)==0)
                        break;
        if(i==n)
                printf("\nFile Not Found");
        else
        {
                printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED");
                printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob);
                for(j=0;j<ft[i].nob;j++)
                        printf("%d, ",ft[i].blocks[j]);
        }
        getch();
}
```

INPUT:

```
Enter no of files   : 2

Enter file 1    : A
Enter no of blocks in file 1      : 4
Enter the blocks of the file 1          : 122394

Enter file 2    : G
Enter no of blocks in file 2      : 5
Enter the blocks of the file 2          :8877665544
Enter the file to be searched  : G
```

OUTPUT:

```
FILE NAME          NO OF BLOCKS                BLOCKS OCCUPIED
G                             5                        88,   77,   66,   55,  44
```

EXPERIMENT 6

6.1 OBJECTIVE

*Write a C program to simulate disk scheduling algorithms
a) FCFS b) SCAN c) C-SCAN

6.2 DESCRIPTION

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

6.3 PROGRAM

6.3.1 FCFS DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int t[20], n, I, j, tohm[20], tot=0;
        float avhm;
        clrscr();
        printf("enter the no.of tracks");
        scanf("%d",&n);
        printf("enter the tracks to be traversed");
        for(i=2;i<n+2;i++)
                scanf("%d",&t*i+);
        for(i=1;i<n+1;i++)
        {
                tohm[i]=t[i+1]-t[i];
                if(tohm[i]<0)
                tohm[i]=tohm[i]*(-1);
         }
         for(i=1;i<n+1;i++)
                tot+=tohm[i];
         avhm=(float)tot/n;
         printf("Tracks traversed\tDifference between tracks\n");
         for(i=1;i<n+1;i++)
                printf("%d\t\t\t%d\n",t*i+,tohm*i+);
         printf("\nAverage header movements:%f",avhm);
         getch();
}
```

INPUT

```
Enter no.of tracks:9
Enter track position:55    58    60     70     18     90     150     160    184
```

OUTPUT

```
Tracks traversed          Difference between tracks
        55                          45
        58                          3
        60                          2
        70                          10
        18                          52
        90                          72
        150                         60
        160                         10
        184                         24

Average header movements:30.888889
```

6.3.2 SCAN DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;

        clrscr();
        printf("enter the no of tracks to be traveresed");
        scanf("%d'",&n);
        printf("enter the position of head");
        scanf("%d",&h);
        t[0]=0;t[1]=h;
        printf("enter the tracks");
        for(i=2;i<n+2;i++)
                scanf("%d",&t[i]);
        for(i=0;i<n+2;i++)
        {
                for(j=0;j<(n+2)-i-1;j++)
                {if(t[j]>t[j+1])
                                {
                                        temp=t[j];
                                        t[j]=t[j+1];
                                        t[j+1]=temp;
        }  }  }
        for(i=0;i<n+2;i++)
                if(t[i]==h)
                        j=i;k=i;
        p=0;
        while(t[j]!=0)
        {
                atr[p]=t[j];
                j--;
                p++;
        }
        atr[p]=t[j];
        for(p=k+1;p<n+2;p++,k++)
                atr[p]=t[k+1];
        for(j=0;j<n+1;j++)
        {
                if(atr[j]>atr[j+1])
                        d[j]=atr[j]-atr[j+1];
                else
                        d[j]=atr[j+1]-atr[j];
                sum+=d[j];
        }
        printf("\nAverage header movements:%f",(float)sum/n); getch();
}
```

INPUT

```
Enter the track position : 55        58        60        70        18        90        150        160        184
Enter starting position : 100
```

OUTPUT

```
 Tracks traversed  Difference Between tracks
             150                    50
             160                    10
             184                    24
             18                     240
             55                     37
             58                     3
             60                     2
             70                     10
             90                     20
Average seek time : 35.7777779
```

6.3.3 C-SCAN DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
        clrscr();
        printf("enter the no of tracks to be traveresed");
        scanf("%d'",&n);
        printf("enter the position of head");
        scanf("%d",&h);
        t[0]=0;t[1]=h;
        printf("enter total tracks");
        scanf("%d",&tot);
        t[2]=tot-1;
        printf("enter the tracks");
        for(i=3;i<=n+2;i++)
                scanf("%d",&t[i]);
        for(i=0;i<=n+2;i++)
                for(j=0;j<=(n+2)-i-1;j++)
                        if(t[j]>t[j+1])
                        {
                                temp=t[j];
                                t[j]=t[j+1];
                                t[j+1]=temp;
                        }
        for(i=0;i<=n+2;i++)
                if(t[i]==h)
                        j=i;break;
        p=0;
        while(t[j]!=tot-1)
        {
                atr[p]=t[j];
                j++;
                p++;
        }
        atr[p]=t[j];
        p++;
        i=0;
        while(p!=(n+3) && t[i]!=t[h])
        {
                atr[p]=t[i];
                i++;
                p++;
        }
```

```
            for(j=0;j<n+2;j++)
            {
                        if(atr[j]>atr[j+1])
                                    d[j]=atr[j]-atr[j+1];
                        else
                                    d[j]=atr[j+1]-atr[j];
                        sum+=d[j];

            }
            printf("total header movements%d",sum);
            printf("avg is %f",(float)sum/n);
            getch();
 }
```

INPUT

```
Enter the track position : 55      58      60      70      18      90      150      160      184
Enter starting position : 100
```

OUTPUT

```
 Tracks traversed  Difference Between tracks
       150              50
       160              10
       184              24
       18               240
       55               37
       58               3
       60               2
       70               10
       90               20
Average seek time : 35.7777779
```

EXPERIMENT 7

7.1 OBJECTIVE

*****Write a C program to simulate the following contiguous memory allocation techniques

a) Worst-fit b) Best-fit c) First-fit

7.2 DESCRIPTION

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

7.3 PROGRAM

7.3.1 *FIRST-FIT*

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

void 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("\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]);
        getch();
}
```

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
```

7.3.2 BEST-FIT

```
#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();
}
```

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
```

7.3.3 WORST-FIT

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

void 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]);
                getch();
        }
```

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
```

EXPERIMENT 8

8.1 OBJECTIVE

, Write programs using the I/0 system calls of UNIX/LINUX operating system (open,read,write,close,fnctl,seek,stat,,opendir readdir).

8.2 DESCRIPTION

What is the File Descripter?

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

File Descriptor table: File descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in operating system for each process.

Read from stdin => read from fd 0 : Whenever we write any character from keyboard, it read from stdin through fd 0 and save to file named /dev/tty.

Write to stdout => write to fd 1 : Whenever we see any output to the video screen, it's from the file named /dev/tty and written to stdout in screen through fd 1.

Write to stderr => write to fd 2 : We see any error to the video screen, it is also from that file write to stderr in screen through fd 2.

Basically there are total 5 types of I/O system calls

```
// C program to illustrate
// open system call
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{
  // if file does not have in directory
  // then file foo.txt is created.
  int fd = open("foo.txt", O_RDONLY | O_CREAT);

  printf("fd = %d/n", fd);

  if (fd ==-1)
  {
    // print which type of error have in a code
    printf("Error Number % d\n", errno);

    // print program detail "Success or failure"
    perror("Program");
  }
  return 0;
}
    // C program to illustrate close system Call
    #include<stdio.h>
    #include <fcntl.h>
    int main()
    {
        int fd1 = open("foo.txt", O_RDONLY);
        if (fd1 < 0)
        {
            perror("c1");
            exit(1);
        }
        printf("opened the fd = % d\n", fd1);

        // Using close system Call

    if (close(fd1) < 0)
        {
            perror("c1");
            exit(1);
        }
        printf("closed the fd.\n");
    }
```

Output:

```
opened the fd = 3

closed the fd.
```

```
// C program to illustrate
// read system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
  int fd, sz;
  char *c = (char *) calloc(100, sizeof(char));

  fd = open("foo.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = read(fd, c, 10);
  printf("called read(% d, c, 10).  returned that"
         " %d bytes  were read.\n", fd, sz);
  c[sz] = '\0';
  printf("Those bytes are as follows: % s\n", c);
}
```

Output:

```
called read(3, c, 10).  returned that 10 bytes  were read.

Those bytes are as follows: 0 0 0 foo.
```

```
// C program to illustrate
// read system Call
#include<stdio.h>
#include<fcntl.h>

int main()
{
    char c;
    int fd1 = Open("foobar.txt", O_RDONLY, 0);
    int fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = % c\n", c);
    exit(0);
}
```

Output:

```
c = f
```

```
// C program to illustrate
// write system Call
#include<stdio.h>
#include <fcntl.h>
main()
{
  int sz;

  int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd < 0)
  {
     perror("r1");
     exit(1);
```

```
    }
sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));

  printf("called write(% d, \"hello geeks\\n\", %d)."
    " It returned %d\n", fd, strlen("hello geeks\n"), sz);

  close(fd);
}
```

Output:

```
called write(3, "hello geeks\n", 12).  it returned 11
```

```
// C Program to demonstrate the use of fseek()
#include <stdio.h>

int main()
{
    FILE *fp;
    fp = fopen("test.txt", "r");

    // Moving pointer to end
    fseek(fp, 0, SEEK_END);

    // Printing position of pointer
    printf("%ld", ftell(fp));

    return 0;
}
```

Output:81

C Program to list all files and sub-directories in a directory

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

int main(void)
{
    struct dirent *de;  // Pointer for directory entry

    // opendir() returns a pointer of DIR type.
    DIR *dr = opendir(".");

    if (dr == NULL)  // opendir returns NULL if couldn't open directory
    {
        printf("Could not open current directory" );
        return 0;
    }

    // Refer http://pubs.opengroup.org/onlinepubs/7990989775/xsh/readdir.html
    // for readdir()
    while ((de = readdir(dr)) != NULL)
            printf("%s\n", de->d_name);

    closedir(dr);
    return 0;
}
```

Output:

```
               All files and subdirectories of current directory
```

Using **readdir()** system call function is as follows.

```
/*  ** Made by BadproG.com */
```

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>
int main(int c, char *v[]) {
    DIR *myDirectory;
    struct dirent *myFile;

    if (c == 2) {
        myDirectory = opendir(v[1]);
        if (myDirectory) {
            puts("OK the directory is opened, let's see its files:");
            while ((myFile = readdir(myDirectory)))
                printf("%s\n", myFile->d_name);
            /*
             ** closedir
             */
            if (closedir(myDirectory) == 0)
                puts("The directory is now closed.");
            else
                puts("The directory can not be closed.");
        } else if (errno == ENOENT)
            puts("This directory does not exist.");
        else if (errno == ENOTDIR)
            puts("This file is not a directory.");
        else if (errno == EACCES)
            puts("You do not have the right to open this folder.");
        else
            puts("That's a new error, check the manual.");
    } else
        puts("Sorry we need exactly 2 arguments.");
    return (0);
}
```

Let's compile this code:

```
$ gcc main.c
```

Then let's execute it:

```
$ ./a.out hello
```

All files in the *hello* directory appear and we finish by closing the directory stream with the closedir() system call function.

EXPERIMENT 9

9.1 OBJECTIVE

a) Write a C program to simulate producer-consumer problem using semaphores.

b) Write a C program to simulate the concept of Dining –Philosophers problem.

9.2 DESCRIPTION

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

The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

9.3 PROGRAM

9.3.1. Producer-Consumer Problem

```
#include<stdio.h>
void main()
{
        int buffer[10], bufsize, in, out, produce, consume, choice=0;
        in = 0;
        out = 0;
        bufsize = 10;
        while(choice !=3)
        {
                printf(“\n1. Produce \t 2. Consume \t3. Exit”);
                printf(“\nEnter your choice: ”);
                scanf(“%d”, &choice);
                switch(choice)    {
                        case 1:  if((in+1)%bufsize==out)
                                        printf(“\nBuffer is Full”);
                                 else
                                 {
                                        printf(“\nEnter the value: “);
                                        scanf(“%d”, &produce);
                                        buffer[in] = produce;
                                        in = (in+1)%bufsize;
                                 }
                                 Break;
                        case 2:  if(in == out)
                                        printf(“\nBuffer is Empty”);
                                 else
                                 {
                                        consume = buffer[out];
                                        printf(“\nThe consumed value is %d”, consume);
                                        out = (out+1)%bufsize;
                                 }
```

```
                                        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
```

9.3.2. Dining-Philosophers problem.

PROGRAM

```
int tph, philname[20], status[20], howhung, hu[20], cho;
main()
{
        int i;
        clrscr();
        printf("\n\nDINING PHILOSOPHER PROBLEM");
        printf("\nEnter the total no. of philosophers: ");
        scanf("%d",&tph);
        for(i=0;i<tph;i++)
        {
                philname[i] = (i+1);
                status[i]=1;
        }
        printf("How many are hungry : ");
        scanf("%d", &howhung);
        if(howhung==tph)
        {
                printf("\nAll are hungry..\nDead lock stage will occur");
                printf("\nExiting..");
        }
        else
        {
                for(i=0;i<howhung;i++)
                {
                        printf("Enter philosopher %d position: ",(i+1));
                        scanf("%d", &hu[i]);
                        status[hu[i]]=2;
                }
                do
                {
                        printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your choice:");
                        scanf("%d", &cho);
                        switch(cho)
                        {
                                case 1:  one();
                                         break;
                                case 2:  two();
                                         break;
                                case 3:  exit(0);
                                default: printf("\nInvalid option..");
                        }
                }while(1);
        }
}
one()
{
        int pos=0, x, i;
        printf("\nAllow one philosopher to eat at any time\n");
        for(i=0;i<howhung; i++, pos++)
        {
                printf("\nP %d is granted to eat", philname[hu[pos]]);
                for(x=pos;x<howhung;x++)
                        printf("\nP %d is waiting", philname[hu[x]]);
        }
}
two()
{
        int i, j, s=0, t, r, x;
        printf("\n Allow two philosophers to eat at same time\n");
        for(i=0;i<howhung;i++)
        {
                for(j=i+1;j<howhung;j++)
                {
                        if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
                        {
                                printf("\n\ncombination %d \n", (s+1));
                                t=hu[i];
                                r=hu[j];
                                s++;
                                printf("\nP %d and P %d are granted to eat", philname[hu[i]],
                                                                        philname[hu[j]]);
                                for(x=0;x<howhung;x++)
                                {
                                        if((hu[x]!=t)&&(hu[x]!=r))
                                        printf("\nP %d is waiting", philname[hu[x]]);
                                }
                        }
                }
        }
}
```

INPUT

```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry : 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5
```

OUTPUT

```
1.One can eat at a time 2.Two can eat at a time 3.Exit
Enter your choice: 1

Allow one philosopher to eat at any time
P 3 is granted to eat
P 3 is waiting
P 5 is waiting
P 0 is waiting
P 5 is granted to eat
P 5 is waiting
P 0 is waiting
P 0 is granted to eat
P 0 is waiting
```

```
1.One can eat at a time 2.Two can eat at a time 3.Exit
Enter your choice: 2

 Allow two philosophers to eat at same time
combination 1
P 3 and P 5 are granted to eat
P 0 is waiting

combination 2
P 3 and P 0 are granted to eat
P 5 is waiting

combination 3
P 5 and P 0 are granted to eat
P 3 is waiting

1.One can eat at a time 2.Two can eat at a time 3.Exit
Enter your choice: 3
```

EXPERIMENT 10

10.1 OBJECTIVE

Write a C program to illustrate the IPC mechanisms.

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

9.2 DESCRIPTION

Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).

- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a ***"virtual file"***.
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "virtual file" or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.

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.

creating a FIFO file: In order to create a FIFO file, a function calls i.e. mkfifo is used.

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

mkfifo() makes a FIFO special file with name ***pathname***. Here ***mode*** specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

Using FIFO: As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. *open*, *read*, *write*, *close*.

Inter Process Communication

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**. New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an

identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

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

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

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

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

```
/ C program to illustrate
// pipe system call in C
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
    char inbuf[MSGSIZE];
    int p[2], i;

    if (pipe(p) < 0)
        exit(1);

    /* continued */
    /* write pipe */

    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);

    for (i = 0; i < 3; i++) {
        /* read pipe */
        read(p[0], inbuf, MSGSIZE);
        printf("% s\n", inbuf);
    }
    return 0;
}
```

Output:

```
hello, world #1

hello, world #2
he
```

```
// C program to illustrate
// pipe system call in C
// shared by Parent and Child
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
    char inbuf[MSGSIZE];
    int p[2], pid, nbytes;

    if (pipe(p) < 0)
        exit(1);

    /* continued */
    if ((pid = fork()) > 0) {
        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);
```

```
        // Adding this line will
        // not hang the program
        // close(p[1]);
        wait(NULL);
    }

    else {
        // Adding this line will
        // not hang the program
        // close(p[1]);
        while ((nbytes = read(p[0], inbuf, MSGSIZE)) > 0)
            printf("% s\n", inbuf);
        if (nbytes != 0)
            exit(2);
        printf("Finished reading\n");
    }
    return 0;
}
```

Output:

```
hello world, #1

hello world, #2

hello world, #3

(hangs)         //program does not terminate but hangs
```

```
/ C program to implement one side of FIFO
// This side writes first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];
    while (1)
    {
        // Open FIFO for write only
        fd = open(myfifo, O_WRONLY);

        // Take an input arr2ing from user.
        // 80 is maximum length
        fgets(arr2, 80, stdin);
// Write the input arr2ing on FIFO
        // and close it
        write(fd, arr2, strlen(arr2)+1);
        close(fd);

        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);
```

```
        // Read from FIFO
        read(fd, arr1, sizeof(arr1));

        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd);
    }
    return 0;
}
```

Output: Run the two programs simultaneously on two terminals.

Program-2

```
// C program to implement one side of FIFO
// This side reads first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd1;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)
    // mkfifo(<pathname>,<permission>)
    mkfifo(myfifo, 0666);

    char str1[80], str2[80];
    while (1)
    {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
read(fd1, str1, 80);

        // Print the read string and close
        printf("User1: %s\n", str1);
        close(fd1);

        // Now open in write mode and write
        // string taken from user.
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

```
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;
```

```
int main()
```

```
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
message.mesg_type = 1;

    printf("Write Data : ");
    gets(message.mesg_text);

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

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

    return 0;
}
```

MESSAGE QUEUE FOR READER PROCESS

```
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);
// display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

SHARED MEMORY FOR WRITER PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

SHARED MEMORY FOR READER PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```