2019

Everest
Engineering
College

Er. Lali Manandhar

[AOS LAB MANUAL]

Table of Contents

APPLIED OPERATING SYSTEMS LABORATORY

OBJECTIVE:

This lab complements the applied 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.

OUTCOMES: Upon the completion of Applied 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.

LAB 1

CPU SCHEDULING ALGORITHMS

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;
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();
  }
```

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;
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();
}
```

1.3.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
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();
}
```

1.3.4 PRIORITY CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
float wtavg, tatavg;
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();
}
```

1.4 Output:

Screenshots of the outputs from the above programs.

1.5 Discussions.

1.6 Conclusion.

LAB 2

PROCESS SYNCHRONIZATION

2.1 OBJECTIVE

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

*Write a C program to simulate the concept of Dining-Philosophers problem.

2.2 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.

2.3 PROGRAM OF PRODUCER-CONSUMER

```
#include<stdio.h>
#include<conio.h>
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("\n Buffer is Empty");
        else
        {
                consume = buffer[out];
                printf("\n The consumed value is %d", consume);
                out = (out+1)%bufsize;
        }
        break;
        }
```

```
}
getch();
}
```

2.4 PROGRAM OF DINNING-PHILOSOPHER PROBLEM

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void one();
void two();
int tph, philname[20], status[20], howhung, hu[20], cho;
main()
{
int i;
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("\n1.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);
}

}
void 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]]);
}
}
void 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]]);
                        }
                }
        }
}
}
```

2.5 Output:

Screenshots of the outputs from the above programs.

2.6 Discussions.

2.7 Conclusion.

LAB 3

DEADLOCK

3.1 OBJECTIVE

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

3.2 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.

3.3 PROGRAM OF BANKER'S ALGORITHM

```
#include<stdio.h>
#include<conio.h>
struct file
{
        int all[10];
        int max[10];
        int need[10];
        int flag;
};
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];
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();
}
```

3.4 Output:

Screenshots of the outputs from the above programs.

3.5 Discussions.

3.6 Conclusion.

LAB 4

FRAGMENTATION

4.1 OBJECTIVE

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

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

4.2 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.

4.3 PROGRAM

4.3.1 FIRST-FIT

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

getch();
}
```

4.3.2 BEST-FIT

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

4.3.3 WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
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 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();
}
```

4.4 Output:

Screenshots of the outputs from the above programs.

4.5 Discussions.

4.6 Conclusion.

LAB 5

PAGE REPLACEMENT ALGORITHM

5.1 OBJECTIVE

Write a C program to simulate page replacement algorithms

a) FIFO b) LRU c) LFU d) a) Optimal

5.2 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. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. 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.

5.3 PROGRAM

5.3.1 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;
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();
 }
```

5.3.2 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;
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();
}
```

5.3.3 LFU PAGE REPLACEMENT ALGORITHM

```
#include<stdio.h>
#include<conio.h>
 main()
 {
 int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0;
printf("\nEnter number of page references -- ");
scanf("%d",&m);
printf("\nEnter the reference string -- ");
for(i=0;i<m;i++)
        scanf("%d",&rs[i]);
printf("\nEnter the available no. of frames -- ");
scanf("%d",&f);
 for(i=0;i<f;i++)
 {
        cntr[i]=0;
         a[i]=-1;
}
```

```
printf("\nThe Page Replacement Process is : \n");
for(i=0;i<m;i++)
{
        for(j=0;j<f;j++)
                if(rs[i]==a[j])
                {
                        cntr[j]++;
                        break;
                }
                if(j==f)
                {
                        min = 0;
                        for(k=1;k<f;k++)
                        if(cntr[k]<cntr[min])
                                min=k;
                        a[min]=rs[i];
                        cntr[min]=1;
                        pf++;
                }
                printf("\n");
                for(j=0;j<f;j++)
                        printf("\t%d",a[j]);
                                if(j==f)
                                        printf("\tPF No. %d",pf);
}
printf("\n\n Total number of page faults -- %d",pf);
getch();
}
```

5.3.4 OPTIMAL PAGE REPLACEMENT ALGORITHM

```
#include<stdio.h>
#include<conio.h>
int n;
int findmax(int a[]);
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;
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;
}
```

5.4 Output:

Screenshots of the outputs from the above programs.

5.5 Discussions.

5.6 Conclusion.

LAB 6

DISK SCHEDULING

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<bits/stdc++.h>
using namespace std;
int main()
{
   int i,j,k,n,m,sum=0,x,y,h;
   cout<<"Enter the size of disk\n";
   cin>>m;
   cout<<"Enter number of requests\n";
   cin>>n;
   cout<<"Enter the requests\n";
   // creating an array of size n
   vector <int> a(n);
   for(i=0;i<n;i++)
```

```
        {
      cin>>a[i];
   }
   for(i=0;i<n;i++)
        {
      if(a[i]>m)
                {
         cout<<"Error, Unknown position "<<a[i]<<"\n";
         return 0;
      }
   }
   cout<<"Enter the head position\n";
   cin>>h;

   // head will be at h at the starting
   int temp=h;
   cout<<temp;
   for(i=0;i<n;i++)
        {
      cout<<" -> "<<a[i]<<" ";
      // calculating the difference for the head movement
      sum+=abs(a[i]-temp);
      // head is now at the next I/O request
      temp=a[i];
   }
   cout<<"\n";
   cout<<"Total head movements = "<< sum<<"\n";
   cout<<"Average head movement = "<<(float)sum/n<<"\n";
   return 0;
}
```

6.3.2 SCAN DISK SCHEDULING ALGORITHM

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
   int i,j,k,n,m,sum=0,x,y,h;
   cout<<"Enter the size of disk\n";
```

```
cin>>m;
cout<<"Enter number of requests\n";
cin>>n;
cout<<"Enter the requests\n";
vector <int> a(n),b;
for(i=0;i<n;i++)
        {
   cin>>a[i];
}
for(i=0;i<n;i++)
        {
   if(a[i]>m)
                {
      cout<<"Error, Unknown position "<<a[i]<<"\n";
      return 0;
   }
}
cout<<"Enter the head position\n";
cin>>h;
int temp=h;
a.push_back(h);
a.push_back(m);
sort(a.begin(),a.end());
for(i=0;i<a.size();i++)
        {
   if(h==a[i])
      break;
}
k=i;
if(k<n/2)
        {
   for(i=k;i<a.size();i++)
                {
      b.push_back(a[i]);
   }
   for(i=k-1;i>=0;i--)
                {
      b.push_back(a[i]);
```

```
    }
  }
  else
       {
    for(i=k;i>=0;i--)
                {
      b.push_back(a[i]);
    }
    for(i=k+1;i<a.size();i++)
                {
      b.push_back(a[i]);
    }
  }
  temp=b[0];
  cout<<temp;
  for(i=1;i<b.size();i++)
       {
    cout<<" -> "<<b[i];
    sum+=abs(b[i]-temp);
    temp=b[i];
  }
  cout<<"\n";
  cout<<"Total head movements = "<< sum<<"\n";
  cout<<"Average head movement = "<<(float)sum/n<<"\n";
  return 0;
}
```

6.3.3 C-SCAN DISK SCHEDULING ALGORITHM

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
  int i,j,k,n,m,sum=0,x,y,h;
  cout<<"Enter the size of disk\n";
  cin>>m;
  cout<<"Enter number of requests\n";
  cin>>n;
  cout<<"Enter the requests\n";
```

```
vector <int> a(n),b;
for(i=0;i<n;i++)
        {
   cin>>a[i];
}
for(i=0;i<n;i++)
        {
   if(a[i]>m)
                {
      cout<<"Error, Unknown position "<<a[i]<<"\n";
      return 0;
   }
}
cout<<"Enter the head position\n";
cin>>h;
int temp=h;
a.push_back(h);
a.push_back(m);
sort(a.begin(),a.end());
for(i=0;i<a.size();i++)
        {
   if(h==a[i])
      break;
}
k=i;
if(k<n/2)
        {
   for(i=k;i<a.size();i++)
                {
      b.push_back(a[i]);
   }
   for(i=k-1;i>=0;i--)
                {
      b.push_back(a[i]);
   }
}
else
        {
```

```
    for(i=k;i>=0;i--)
                {
      b.push_back(a[i]);
    }
    for(i=k+1;i<a.size();i++)
                {
      b.push_back(a[i]);
    }
  }
  temp=b[0];
  cout<<temp;
  for(i=1;i<b.size();i++)
       {
    cout<<" -> "<<b[i];
    sum+=abs(b[i]-temp);
    temp=b[i];
  }
  cout<<"\n";
  cout<<"Total head movements = "<< sum<<"\n";
  cout<<"Average head movement = "<<(float)sum/n<<"\n";
  return 0;
}
```

6.4 Output:

Screenshots of the outputs from the above programs.

6.5 Discussions.

6.6 Conclusion.

LAB 7

FILE ORGANIZATION TECHNIQUES

7.1 OBJECTIVE

Write a C program to simulate the following file organization techniques

a) Single level directory b) Two level directory

7.2 DESCRIPTION

The directory structure is the organization of files into a hierarchy of folders. In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory. In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched. This effectively solves the name collision problem and isolates users from one another. Hierarchical directory structure allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A directory (or subdirectory) contains a set of files or subdirectories.

7.3 PROGRAM

7.3.1 SINGLE LEVEL DIRECTORY ORGANIZATION

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

```
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 \n 4. 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);
                        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);
        }

}
getch();
}
```

7.3.2 TWO LEVEL DIRECTORY ORGANIZATION

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

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

```
                                        {
                                                if(strcmp(f, dir[i].fname[k])==0)
                                                {
                                                        printf("File %s is deleted ",f);

                                                        dir[i].fcnt--;

strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
                                                        goto jmp;
                                                }
                                        }
                                        printf("File %s not found",f);
                                        goto jmp;
                                }
                                }
                                printf("Directory %s not found",d);
                                jmp   : break;

        case 4:
                                printf("\nEnter name of the directory -- ");
                        scanf("%s",d);
                        for(i=0;i<dcnt;i++)
                        {
                                        if(strcmp(d,dir[i].dname)==0)
                                        {
                                          printf("Enter the name of the file --   ");
                                          scanf("%s",f);
                                          for(k=0;k<dir[i].fcnt;k++)
                                          {
                                                        if(strcmp(f, dir[i].fname[k])==0)
                                                        {
                                                                printf("File %s is found ",f);
                                                                goto jmp1;
                                                        }
                                        }
                                        printf("File %s not found",f);
                                        goto jmp1;
                                }
                        }
                        printf("Directory %s not found",d);
                        jmp1:   break;
        case 5:
```

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

}
getch();
}
```

6.4 Output:

Screenshots of the outputs from the above programs.

6.5 Discussions.

6.6 Conclusion.