Experiment No. 1

Implementation of CPU Scheduling:

FIRST COME FIRST SERVE SCHEDULLING (FCFS)

1. First Come First Serve Scheduling

THEORY:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process name and the burst time

Step 4: Set the waiting of the first process as _ 0'and its burst time as its turnaround time

Step 5: for each process in the Ready Q calculate

a). Waiting time (n) = waiting time (n-1) + Burst time (n-1)

b). Turnaround time (n)= waiting time(n)+Burst time(n)

Step 6: Calculate

a). Average waiting time = Total waiting Time / Number of processes

b). Average Turnaround time = Total Turnaround Time / Number of processes

Step 7: Stop the process

SOURCE CODE:

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

printf ("\n Enter the number of processes -- ");
scanf ("%d", &n);
for (i=0;i<n;i++)
{
printf ("\n Enter 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 \t BURST 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 ("\n Average Waiting Time -- %f", wtavg/n);
printf ("\n Average Turnaround Time -- %f", tatavg/n);
getch();
} }
```

INPUT/OUTPUT:

```
Enter the number of processes -- 3

Enter Burst Time for Process 0 -- 2

Enter Burst Time for Process 1 -- 5

Enter Burst Time for Process 2 -- 1
        PROCESS         BURST TIME      WAITING TIME    TURNAROUND TIME

        P0              2               0               2
        P1              5               2               7
        P2              1               7               8
Average Waiting Time -- 3.000000
Average Turnaround Time -- 5.666667
```

RESULT AND CONCLUSION: Thus, the program for CPU Scheduling (FCFS) was executed and verified success.

Experiment No. 2

Implementation of CPU Scheduling:

SHORTEST JOB FIRST (SJF) SCHEDULLING

THEORY:

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according to the shortest Burst time by sorting according to lowest to highest burst time.

Step 5: Set the waiting time of the first process as ‗0' and its turnaround time as its burst time.

Step 6: Sort the processes names based on their Burt time

Step 7: For each process in the ready queue, calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 8: Calculate

c) Average waiting time = Total waiting Time / Number of processes

d) Average Turnaround time = Total Turnaround Time / Number of processes

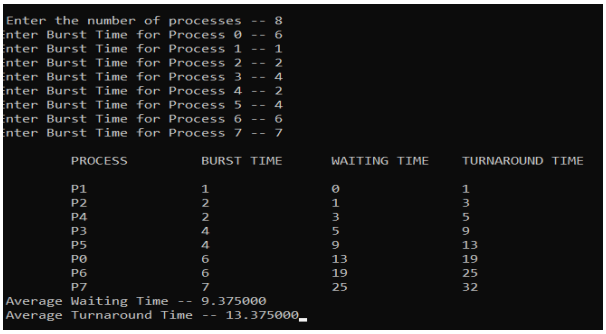
Step 9: Stop the process

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
main () {
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg;
printf ("\n Enter 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 \t BURST 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 ("\n Average Waiting Time -- %f", wtavg/n);
printf ("\n Average Turnaround Time -- %f", tatavg/n);
getch(); }
```

INPUT/OUTPUT:



RESULT AND CONCLUSION: Thus, the program for CPU Scheduling (SJF) was executed and verified success.

Experiment No. 3

Implementation of CPU Scheduling:

ROUND ROBIN (RR) SCHEDULLING

THEORY:

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not, it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assigns the waiting time to the sum of the total times. If it is greater than the burst-time, then subtract the time slot from the actual burst time and increment it by time slot and the loop continue until all the processes are completed.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

a) Waiting time for process (n) = waiting time of process(n-1) + burst time of process (n-1) + the time difference in getting the CPU from process(n-1)

b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process (n).

Step 7: Calculate

c) Average waiting time = Total waiting Time / Number of processes

d) Average Turnaround time = Total Turnaround Time / Number of processes

Step 8: Stop the process.

Source Code:

```
#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 ("\n Enter 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 ("\n The Average Turnaround time is -- %f", att/n);
printf ("\n The Average Waiting time is -- %f ", awt/n);
printf ("\n\t PROCESS \t BURST TIME \t WAITING TIME \t TURNAROUND 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/OUTPUT:

```
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

The Average Turnaround time is -- 15.000000
The Average Waiting time is -- 5.000000
        PROCESS         BURST TIME      WAITING TIME    TURNAROUND TIME
       1        24              6               30
       2        3               3               6
       3        3               6               9
```

Result & Conclusion:

Thus, the program for CPU Scheduling (Round Robin) was executed and verified success.

Experiment No. 4

Implementation of CPU Scheduling:

PRIORITY SCHEDULLING

THEORY:

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround time

Step 6: Arrange the processes based on process priority

Step 7: For each process in the Ready Q calculate

Step 8: for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 9: Calculate

c) Average waiting time = Total waiting Time / Number of processes

d) Average Turnaround time = Total Turnaround Time / Number of processes

Step10: Stop the process.

Source Code:

```
#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 ("\n PROCESS\t\t PRIORITY \t BURST TIME \t WAITING TIME \t TURNAROUND 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 ("\n Average Waiting Time is --- %f",wtavg/n);
printf ("\n Average Turnaround Time is --- %f",tatavg/n);
getch(); }
```

INPUT/OUTPUT:

```
Enter the number of processes --- 2
Enter the Burst Time & Priority of Process 0 --- 4
5
Enter the Burst Time & Priority of Process 1 --- 2
6

 PROCESS                PRIORITY        BURST TIME      WAITING TIME    TURNAROUNDTIME
0                       5                       4               0               4
1                       6                       2               4               6
 Average Waiting Time is --- 2.000000
 Average Turnaround Time is --- 5.000000
Process returned 13 (0xD)   execution time : 8.921 s
Press any key to continue.
```

Result & Conclusion:

Thus, the program for CPU Scheduling (FCFS, SJF, Round Robin and Priority) was executed and verified successfully.

Experiment No. 5

Write a C program to simulate Banker's Algorithm for the purpose of Deadlock Avoidance.

THRORY:

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.

Algorithm:

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

Source code:

```
#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];

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();
}
```

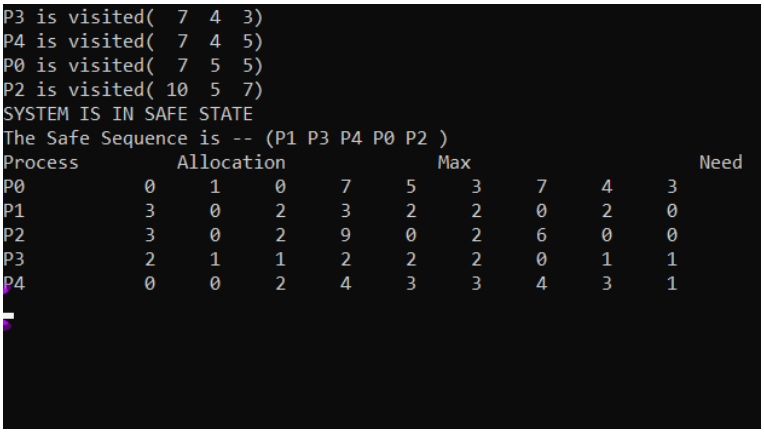
INPUT/OUTPUT:

```
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

P1 is visited(  5  3  2)
```



Result & Conclusion:

Thus, the program for Banker's Algorithm for Deadlock Avoidance was executed and verified successfully.

Experiment No. 6

To Write a C program to simulate Producer-Consumer Problem using Semaphores.

THRORY:

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and 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. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the required variables.

Step 3: Initialize the buffer size and get maximum item you want to produce.

Step 4: Get the option, which you want to do either producer, consumer or exit from the operation.

Step 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size.

Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.

Step 7: If you select exit come out of the program.

Step 8: Stop the program.

Source Code:

```
#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("\n 1. Produce \t 2. Consume \t 3. Exit");
 printf("\n Enter your choice: ");
scanf("%d",&choice); switch(choice)
 {
case 1: if((in+1)%bufsize==out)
printf("\n Buffer is Full");
else {
printf("\n Enter 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;
}
 }
}
```

Result & Conclusion:

Thus, the program for Producer-Consumer Problem using Semaphore was executed and verified successfully.

Experiment No. 7

Implement the following Page Replacement Algorithms:

FIRST IN FIRST OUT (FIFO) ALGORITHM

DESCRIPTION:

The simpler page replacement algorithm is a FIFO algorithm. 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. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue when a page is brought into memory; we insert it at the tail of the queue.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		4	0	0	0	3	3			3	2			2	2	1

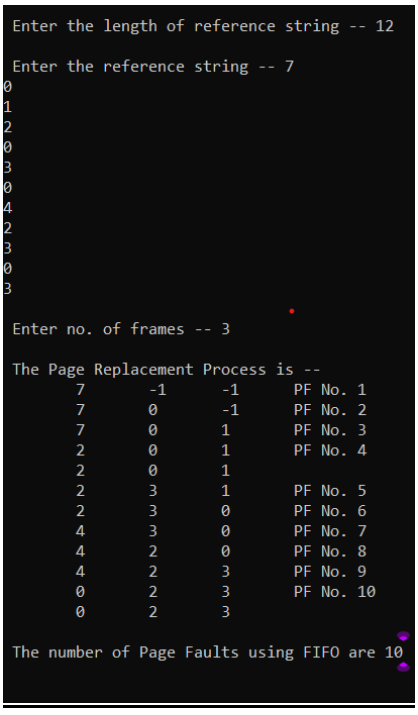
Source Code:

```
#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();
}
```

OUTPUT:



Result & Conclusion:

Thus, the program for Page Replacement Algorithms (FIFO) was executed and verified succe

Experiment No. 8

Implement the following Page Replacement Algorithms:

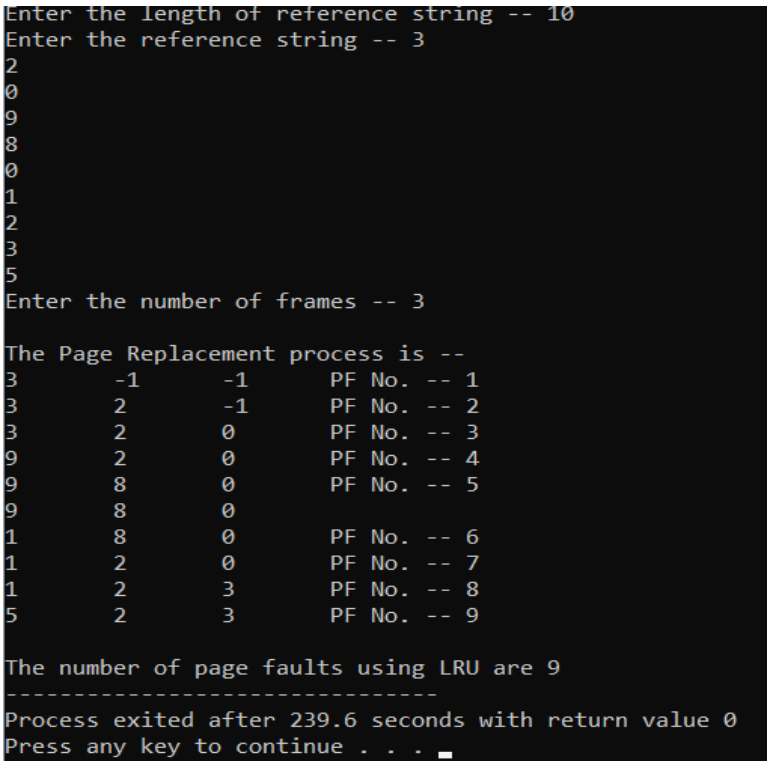
LEAST RECENTLY USED (LRU) ALGORITHM

Source Code:

```
#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();
}
```

INPUT/OUTPUT:



<u>Result & Conclusion</u>:

Thus, the program for Page Replacement Algorithms (LRU) was executed and verified successfully.

Experiment No. 9

Write a C program to simulate Disk Scheduling Algorithms:

FCFS DISK SCHEDULING ALGORITHM

Source Code:

```
#include<stdio.h>
main()
{
 int t[20], n, I, j, tohm[20], tot=0; float avhm;
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;i++)
{
 tohm[i]=t[i+1]-t[i];
if(tohm[i]<0)
tohm[i]=tohm[i]*(-1);
}
for(i=1;i<n;i++)
tot+=tohm[i];
avhm=(float)tot/n;
printf(“Tracks traversed \t Difference between tracks\n”);
 for(i=1;i<n;i++)
printf(“%d\t\t\t%d\n”,t*i+,tohm*i+);
printf("\n Average header movements:%f",avhm);
 getch();
}
```

Experiment No. 10

Write a C program to simulate Disk Scheduling Algorithms:

SCAN DISK SCHEDULING ALGORITHM

Software Required: Turbo C / Borland C

Source Code:

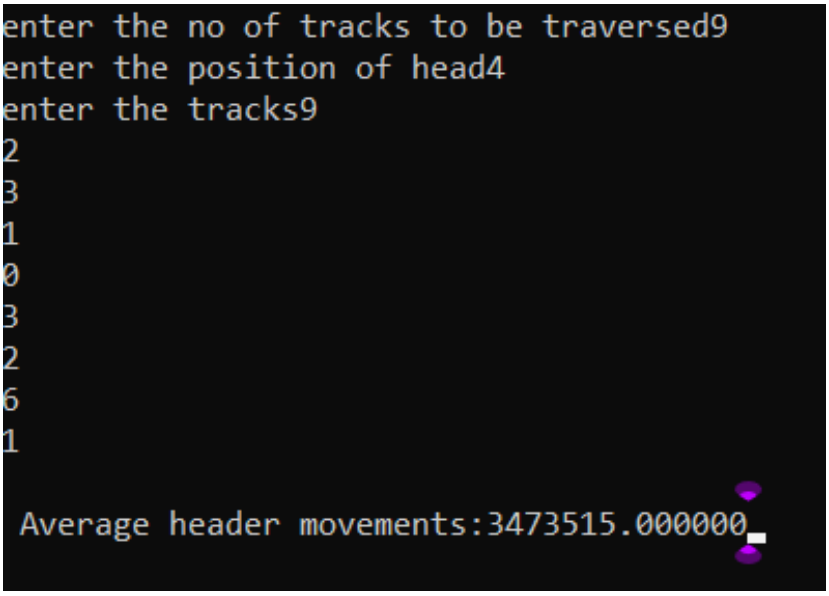
```
#include<stdio.h>
main()
{
int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;

printf("enter the no of tracks to be traversed");
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("\n Average header movements:%f",(float)sum/n);
getch();
}
```

INPUT/OUTPUT:



RESULTS AND CONCLUSION:

Thus, this program to simulate Disk Scheduling Algorithms (SCAN) was successfully

Verified.

Experiment No. 11

Write a C program to simulate Disk Scheduling Algorithms:

C - SCAN DISK SCHEDULING ALGORITHM

Software Required: Turbo C / Borland C

Source Code:

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

printf("enter the no of tracks to be traversed");
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;

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/OUTPUT:

```
enter the no of tracks to be traversed7
enter the position of head5
enter total tracks9
enter the tracks2
0
3
0
2
4
3
total header movements21759647avg is 3108521.250000
```

RESULTS AND CONCLUSION:

Thus, this program to simulate Disk Scheduling Algorithms(C-SCAN) was successfully Verified.

Experiment No. 12

To Write a C program to simulate the following Contiguous Memory Allocation Techniques:

Worst-fit

Source Code:

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.

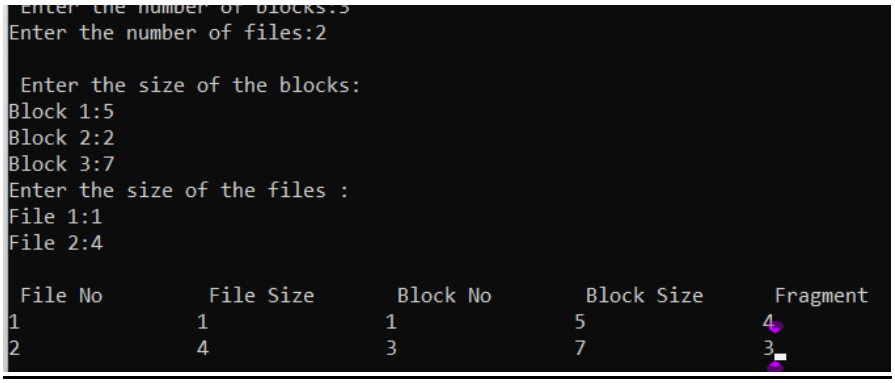
```
#include<stdio.h>
#include<conio.h>
#define max 25
main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
static int bf[max],ff[max];

printf("\n\t Memory Management Scheme - First Fit");
printf("\n Enter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\n Enter the size of the blocks:\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :\n");
```

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

{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}
printf("\n File No \t File Size \t Block No \t Block Size \t Fragment");
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 /OUTPUT :



Result & Conclusion:

Thus, the program for Contiguous Memory Allocation Techniques (Worst-Fit) was executed and verified success.

Experiment No. 13

To Write a C program to simulate the following Contiguous Memory Allocation Techniques:

BEST-FIT

Source Code:

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


printf("\n Enter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\n Enter 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("\n File No \t File Size \t Block No \t Block Size \t Fragment");
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/OUTPUT:

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

 Enter the size of the blocks:
Block 1:5
Block 2:2
Block 3:7
Enter the size of the files:
File 1:1
File 2:4

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

Result & Conclusion:

Thus, the program for Contiguous Memory Allocation Techniques (Best-Fit) was executed and verified success.

Experiment No. 14

To Write a C program to simulate the following Contiguous Memory Allocation Techniques:

FIRST-FIT

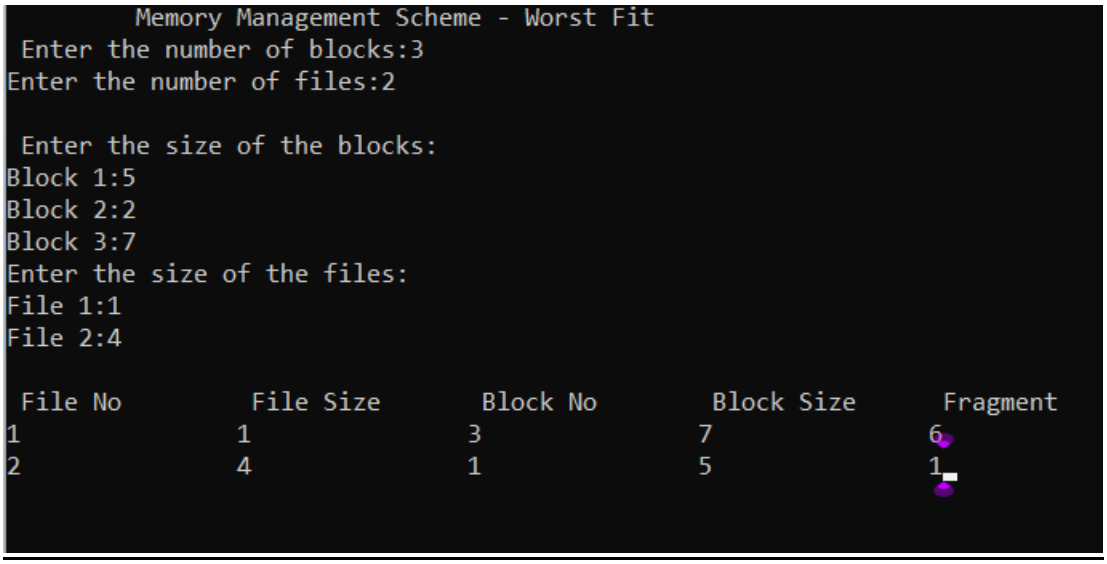
Source Code:

```
#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];
printf("\n\t Memory Management Scheme - Worst Fit");
 printf("\n Enter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\n Enter 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)
{
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("\n File No \t File Size \t Block No \t Block Size \t Fragment");
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/OUTPUT:



Result & Conclusion:

Thus, the program for Contiguous Memory Allocation Techniques (First-Fit) was executed and verified success.