# WEEK-8

**AIM:** Implement non-pre-emptive/pre-emptive CPU scheduling algorithms to find turnaround time and waiting time (minimum 2 from all process scheduling algorithms)

**FISRT COME FISRT SERVE :**

First come First serve scheduling algorithm simply schedules the jobs according to their arrival time.The job which comes first in the ready queue will get the CPU first. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

**PROGRAM:**

```
#include<stdio.h>
#include<stdlib.h>
struct fcfs
{
        int st,ft,at,bt,wt,tat,pr;
}
p[100],temp;
int main()
{
        float avg_wt,avg_tat;
        int n,i,j;
        avg_wt=avg_tat=0;
        printf("Enter number of processes:\n");
        scanf("%d",&n);
        printf("Enter the ARRIVAL TIME of process %d:\n",i+1);
        for(i=0;i<n;i++)
        {
                p[i].pr=i+1;
                scanf("%d",&p[i].at);
         }
        printf("Enter the BURST TIME of process %d:\n",i+1);
        for(i=0;i<n;i++)
        {
                scanf("%d",&p[i].bt);
        }
        for(i=0;i<n-1;i++)
        {
                for(j=1;j<n-1;j++)
                {
                        if(p[j].at>p[j+1].at)
                {
                        temp=p[j];
                        p[j]=p[j+1];
                        p[j+1]=temp;
                }
                }
        }
printf("PROCESS\t AT\t BT\t ST\t FT\t WT\t TAT\n");
p[0].st=p[0].at;
for(i=1;i<n;i++)
{
        if(p[i].at>p[i-1].st+p[i-1].bt)
        {
                p[i].st=p[i].at;
        }
        else
        {
                p[i].st=p[i-1].st+p[i-1].bt;
        }
}
for(i=0;i<n;i++)
{
        p[i].ft=p[i].st+p[i].bt;
        p[i].wt=p[i].st-p[i].at;
        p[i].tat=p[i].wt+p[i].bt;
}
for(i=0;i<n;i++)
{
        printf("%d\t %d\t %d\t %d\t %d\t %d\t %d\n",p[i].pr,p[i].at,p[i].
bt,p[i].st,p[i].ft,p[i].wt,p[i].tat);
        printf("\n");
}
for(i=0;i<n;i++)
{
        avg_wt=avg_wt+p[i].wt;
        avg_tat=avg_tat+p[i].tat;
}
printf("\n AVERAGE WAITING TIME: %f\n",(avg_wt)/n);
printf("\n AVERAGE TURN AROUND TIME :%f\n",(avg_tat)/n);
return 0;
}
```

**OUTPUT:**

```
Enter number of processes:
```

```
3
Enter the ARRIVAL TIME of process 1:
0
0
0
Enter the BURST TIME of process 4:
24
3
3
PROCESS  AT    BT    ST    FT    WT    TAT
1         0    24     0    24     0    24

2         0     3    24    27    24    27

3         0     3    27    30    27    30

AVERAGE WAITING TIME: 17.000000
AVERAGE TURN AROUND TIME :27.000000
```

**SHORTEST JOB FIRST:**

Shortest job first scheduling algorithm, schedules the processes according to theri burst time.The process with the lowest burst time, among the list of available processes in the ready queue,is going to be scheduled next.it is very difficult to predict the burst time needed for a process. It requires maximum throughput and gives minimum average waiting time and turn around time.

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
main()
{
 int p[20],at[20], bt[20], wt[20], tat[20], i, k, n, temp;
 float avgwt,avgtat;
 printf("\nEnter the number of processes :");
 scanf("%d", &n);
 for(i=0;i<n;i++)
 {
     printf("Enter the ARRIVAL TIME for process[%d] :",i);
     scanf("%d",&at[i]);
 }
```

```
    printf("Enter the BURST TIME:",i);
    for(i=0;i<n;i++)
    {
      p[i]=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;
          }
          else
           {
              temp=p[i];
              p[i]=p[k];
              p[k]=temp;
          }
      }
      wt[0] = avgwt= 0;
      tat[0] = avgtat= bt[0];
      for(i=1;i<n;i++)
      {
          wt[i] = wt[i-1] +bt[i-1];
          tat[i] = tat[i-1] +bt[i];
          avgwt = avgwt + wt[i];
          avgtat = avgtat + tat[i];
      }
      printf("\n PROCESS\t AT\t  BT\t  WT\t  TAT\n");
      for(i=0;i<n;i++)
      printf("p[%d]\t\t %d\t %d\t %d\t %d\n", p[i],at[i],bt[i], wt[i], tat[i]);
      printf("\nAVERAGE WAITING TIME: %f", avgwt/n);
      printf("\nAVERAGE TURN AROUND TIME %f", avgtat/n);
      getch();
}
```

**OUTPUT:**

```
Enter the number of processes :4
Enter the ARRIVAL TIME for process[0] :0
Enter the ARRIVAL TIME for process[1] :2
Enter the ARRIVAL TIME for process[2] :1
Enter the ARRIVAL TIME for process[3] :4
Enter the BURST TIME:
6
8
7
3
PROCESS   AT     BT     WT     TAT
   p[2]         0       3        0        3
   p[0]         2       6        3        9
   p[1]         1       7        9        16
   p[3]         4       8        16       24

AVERAGE WAITING TIME: 7.000000
AVERAGE TURN AROUND TIME 13.000000
```

| SNO | TASK | MARKS |
|---|---|---|
| 1 | Observation, Program writing and Execution | /10 |
| 2 | Record Submission | /5 |
| 3 | Viva-voce | /5 |
| 4 | Total | /20 |

# WEEK-9

**AIM:** Implement process synchronization using Semaphores (use any one real time example application)

**SEMAPHORE:**

Semaphore is a very significant technique to manage concurrent processes by using a simple integer value,which is known as Semaphore. Semaphore is simply an integer variable that is shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

**CODE:**

```
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *Func1();
 void *Func2();
 int shared=1;
 sem_t s;
 int main()
 {
 sem_init(&s,0,1);
 pthread_t thread1,thread2;
 pthread_create(&thread1,NULL,Func1,NULL);
 pthread_create(&thread2,NULL,Func2,NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared);
 }
void *Func1()
{
   int x;
   sem_wait(&s);
   x=shared;
   printf("Thread1 reads the value as %d\n",x);
   x++;
   printf("Local updation by Thread1 is : %d\n",x);
   sleep(10);
   shared=x;
   printf("Value of shared variable updated by Thread1
      is: %d\n",shared);
   sem_post(&s);
}
void *Func2()
{
   int y;
   sem_wait(&s);
   y=shared;
   printf("Thread2 reads the value as %d\n",y);
   y--;
   printf("Local updation by Thread2 is: %d\n",y);
   sleep(10);
   shared=y;
   printf("Value of shared variable updated by Thread2
     is: %d\n",shared);
   sem_post(&s);
}
```

**OUTPUT:**

```
Thread1 reads the value as 1
Local updation by Thread1 is : 2
Value of shared variable updated by Thread1 is: 2
Thread2 reads the value as 2
Local updation by Thread2 is: 1
Value of shared variable updated by Thread2 is: 1
Final value of shared is 1
```

| SNO | TASK | MARKS |
|---|---|---|
| 1 | Observation, Program writing and Execution | /10 |
| 2 | Record Submission | /5 |
| 3 | Viva-voce | /5 |
| 4 | Total | /20 |

# WEEK-10

**AIM:** Implement Banker's algorithm for the purpose of Deadlock avoidance

**BANKER`S ALGORITHM:**

The Banker`s alogrithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources,then makes an "S-state"check to test for possible activities ,before deciding whether allocation should be allowed to continue.

**CODE:**

```
#include <stdio.h>
#include <conio.h>
int main()
{
int
      Max[10][10],need[10][10],alloc[10][10],avail[10],completed[10],
      safeSeq[10];
int n,m,i,j,process,count;
count=0;
printf("Enter the no of processes : ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
      completed[i]=0;
}
printf("\n\nEnter the no of resources : ");
scanf("%d",&m);
printf("\n\nEnter the Max Matrix for each process : ");
for(i=0;i<n;i++)
{
      printf("\nFor process %d : ",i+1);
      for(j=0;j<m;j++)
      {
        scanf("%d",&Max[i][j]);
      }
}
printf("\n\nEnter the allocation for each process : ");
for(i=0;i<n;i++)
{
      printf("\nFor process %d : ",i+1);
```

```
        for(j=0;j<m;j++)
        {
          scanf("%d", &alloc[i][j]);
        }
}
printf("\n\nEnter the Available Resources : ");
for(i=0;i<m;i++)
    {
          scanf("%d",&avail[i]);
        }
        for(i=0;i<n;i++)
        {
          for(j=0;j<m;j++)
          {
                need[i][j]=Max[i][j]-alloc[i][j];
          }
        }
do
{
        printf("\n Max matrix:\tAllocation matrix:\n");
        for(i=0;i<n;i++)
        {
          for(j=0;j<m;j++)
          {
                printf("%d  ",Max[i][j]);
          printf("\t\t");
            }
          for(j=0;j<m;j++)
          {
                printf("%d  ",alloc[i][j]);
          printf("\n");
            }
        }
        process=-1;
        for(i=0;i<n;i++)
        {
          if(completed[i]==0)
          {
                process=i;
                for(j=0;j<m;j++)
                {
                        if(avail[j]<need[i][j])
                        {
```

```
                                    process=-1;
                                    break;
                            }
                    }
            }
            if(process!=-1)
                    break;
        }
        if(process!=-1)
        {
          printf("\nProcess %d runs to completion!",process + 1);
          safeSeq[count]=process+1;
          count++;
          for(j=0;j<m;j++)
          {
                  avail[j]+=alloc[process][j];
                  alloc[process][j]=0;
                  Max[process][j]=0;
                  completed[process]=1;
          }
        }
}while(count!=n&&process!=-1);
if(count==n)
{
        printf("\nThe system is in a safe state!!\n");
        printf("Safe Sequence : < ");
        for(i=0;i<n;i++)
                printf("%d  ",safeSeq[i]);
        printf(">\n");
}
else
        printf("\nThe system is in an unsafe state!!");
getch();
}
```

**OUTPUT:**

```
Enter the no of processes : 5
Enter the no of resources : 3
Enter the Max Matrix for each process :
For process 1 : 7 5 3
For process 2 : 3 2 2
For process 3 : 9 0 2
For process 4 : 2 2 2
For process 5 : 4 3 3
Enter the allocation for each process :
For process 1 : 0 1 0
For process 2 : 2 0 0
For process 3 : 3 0 2
For process 4 : 2 1 1
For process 5 : 0 0 2
Enter the Available Resources :
 3
 3
 2
 Max matrix:    Allocation matrix:
7  5  3          0  1  0
3  2  2          2  0  0
9  0  2          3  0  2
2  2  2          2  1  1
4  3  3          0  0  2
Process 2 runs to completion!
 Max matrix:    Allocation matrix:
7  5  3          0  1  0
0  0  0          0  0  0
9  0  2          3  0  2
2  2  2          2  1  1
4  3  3          0  0  2
Process 4 runs to completion!
 Max matrix:    Allocation matrix:
7  5  3          0  1  0
0  0  0          0  0  0
9  0  2          3  0  2
0  0  0          0  0  0
4  3  3          0  0  2
Process 1 runs to completion!
 Max matrix:    Allocation matrix:
0  0  0          0  0  0
```

```
0  0  0              0  0  0
9  0  2              3  0  2
0  0  0              0  0  0
4  3  3              0  0  2
Process 3 runs to completion!
 Max matrix:    Allocation matrix:
0  0  0              0  0  0
0  0  0              0  0  0
0  0  0              0  0  0
0  0  0              0  0  0
4  3  3              0  0  2
Process 5 runs to completion!
The system is in a safe state!!
Safe Sequence : < 2  4  1  3  5  >
```

| SNO | TASK | MARKS |
|---|---|---|
| 1 | Observation, Program writing and Execution | /10 |
| 2 | Record Submission | /5 |
| 3 | Viva-voce | /5 |
| 4 | Total | /20 |

# WEEK-13

**AIM:** Implement Page Replacement algorithms.

**FIRST IN FIRST OUT(FIFO):**

This is the simplest page replacement algorithm.In this algorithm,the operating system keeps track of all pages in the memory in a queue,the oldest page is in the front of the queue. when a page needs to be replaced page in the front of the queue is selected for removal.

**Page References: 1,3,0,3,5,6,3**

**No.of Frames=3**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
|  |  | 0 | 0 | 0 | 0 | 3 |
|  | 3 | 3 | 3 | 3 | 6 | 6 |
| 1 | 1 | 1 | 1 | 5 | 5 | 5 |
| **PF** | **PF** | **PF** | **HIT** | **PF** | **PF** | **PF** |

**Total Page Faults=6**

**CODE:**

```
#include <stdio.h>
int main()
{
int referenceString[10],pageFaults=0,m,n,s,p,f;
printf("\nEnter the number of Pages:\n");
scanf("%d",&p);
printf("\nEnter reference string values:\n");
for(m=0;m<p;m++)
{
   printf("Value No. [%d]:\t",m+1);
   scanf("%d",&referenceString[m]);
}
printf("\n Enter the number of frames:\t");
{
   scanf("%d",&f);
}
int temp[f];
for(m=0;m<f;m++)
{
  temp[m]=-1;
}
for(m=0;m<p;m++)
{
  s=0;
  for(n=0;n<f;n++)
   {
      if(referenceString[m]==temp[n])
        {
           s++;
           pageFaults--;
        }
   }
   pageFaults++;
   if((pageFaults<=f)&&(s==0))
      {
        temp[m]=referenceString[m];
      }
   else if(s==0)
      {
        temp[(pageFaults-1)%f]=referenceString[m];
      }
      printf("\n");
      for(n=0;n<f;n++)
       {
         printf("%d\t",temp[n]);
       }
}
printf("\nTotal Page Faults:\t%d\n",pageFaults);
return 0;
}
```

**OUTPUT:**

```
Enter the number of Pages:
7
Enter reference string values:
Value No. [1]:  1
Value No. [2]:  3
Value No. [3]:  0
Value No. [4]:  3
Value No. [5]:  5
Value No. [6]:  6
Value No. [7]:  3
Enter the number of frames:   3
1       -1      -1
1       3       -1
1       3       0
1       3       0
```

```
5       3       0
5       6       0
5       6       3
Total Page Faults:      6
```

**LEAST RECENTLY USED(LRU):**

Least Recently Used algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference,the least recently used page is not likely.

**Page Reference=7,0,1,2,0,3,0,4,2,3,0,3,2,3**

**No.of Frames=4**

| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
|  |  | 1 | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **PF** | **PF** | **PF** | **PF** | **H** | **PF** | **H** | **PF** | **H** | **H** | **H** | **H** | **H** | **H** |

**Page Faults=6**

**CODE:**

```
#include <stdio.h>
int findLRU(int time[], int n)
{
   int i,minimum=time[0],position=0;
   for (i=1;i<n;++i)
   {
      if (time[i]<minimum)
      {
         minimum=time[i];
         position = i;
      }
   }
   return position;
}
int main()
{
   int
f,p,frames[10],pages[30],counter=0,time[10],flag1,flag2,i,j,position,f
aults = 0;
```

```
printf("Enter number of pages: ");
scanf("%d",&p);
printf("Enter reference string: ");
for(i=0;i<p;++i)
{
  scanf("%d",&pages[i]);
}
printf("Enter number of frames: ");
scanf("%d",&f);
for(i=0;i<f;++i)
{
  frames[i]=-1;
}
for(i=0;i<p;++i)
{
  flag1=flag2=0;
  for(j=0;j<f;++j)
  {
    if(frames[j]==pages[i])
    {
      counter++;
      time[j]=counter;
      flag1=flag2=1;
      break;
    }
  }
  if(flag1==0)
  {
    for(j=0;j<f;++j)
    {
      if(frames[j]==-1)
      {
        counter++;
        faults++;
        frames[j]=pages[i];
        time[j]=counter;
        flag2=1;
        break;
      }
    }
  }
  if(flag2==0)
  {
        position=findLRU(time,f);
        counter++;
        faults++;
        frames[position]=pages[i];
        time[position]=counter;
      }
      printf("\n");

      for(j=0;j<f;++j)
      {
        printf("%d\t",frames[j]);
      }
    }
    printf("\nTotal Page Faults = %d",faults);
    return 0;
}
```

**OUTPUT:**

```
Enter number of pages: 10
Enter reference string: 7 5 9 4 3 7 9 6 2 1
Enter number of frames: 3
7       -1      -1
7       5       -1
7       5       9
4       5       9
4       3       9
4       3       7
9       3       7
9       6       7
9       6       2
1       6       2
Total Page Faults = 10
```

| SNO | TASK | MARKS |
|---|---|---|
| 1 | Observation, Program writing and Execution | /10 |
| 2 | Record Submission | /5 |
| 3 | Viva-voce | /5 |
| 4 | Total | /20 |

# WEEK-11

**AIM:** Implement the MVT and MFT Memory Management techniques

## MULTI-PROGRAMMING WITH FIXED PARTITIONING(MFT):

Multi-programming with fixed partitioning is a continguous memory management technique in which the main memory is divided into fixed sized partitions which can be of equal or unequal size.Whenever we have to allocate a process memory then a free partition that is big enough to hold the process is found.Then the memory is allocated to the process.If there is no free space available then the process waits in the queue to be allocated memory.It is one of the most oldest memory management technique which is easy to implement.

**CODE:**

```
#include<stdio.h>
#include<conio.h>
int main()
{
int ma,bs,nob,ef,n,mp[10],tif=0;
int i,p=0;
printf("Enter the total memory available (in Bytes):");
scanf("%d",&ma);
printf("Enter the block size (in Bytes):");
scanf("%d", &bs);
nob=ma/bs;
ef=ma-nob*bs;
printf("\nEnter the number of processes:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter memory required for process %d (in Bytes):",i+1);
scanf("%d",&mp[i]);
}
printf("\nNo. of Blocks available in memory: %d",nob);
printf("\n\nPROCESS\t MEMORY
        REQUIRED\tALLOCATED\tINTERNAL FRAGMENTATION");
for(i=0;i<n&&p<nob;i++)
{
printf("\n %d\t\t%d",i+1,mp[i]);
if(mp[i]>bs)
printf("\t\tNO\t\t---");
else
{
printf("\t\tYES\t\t%d",bs-mp[i]);
tif = tif + bs-mp[i];
p++;
}
}
if(i<n)
printf("\nMemory is Full, Remaining Processes cannot be
        accomodated");
printf("\n\nTotal Internal Fragmentation is %d",tif);
printf("\nTotal External Fragmentation is %d",ef);
getch();
}
```

**OUTPUT:**

```
Enter the total memory available (in Bytes):1000
Enter the block size (in Bytes):300

Enter the number of processes:5

Enter memory required for process 1 (in Bytes):275
Enter memory required for process 2 (in Bytes):400
Enter memory required for process 3 (in Bytes):290
Enter memory required for process 4 (in Bytes):293
Enter memory required for process 5 (in Bytes):100

No. of Blocks available in memory: 3

PROCESS  MEMORYREQUIRED  ALLOCATED   INTERNAL
                                  FRAGMENTATION
    1          275             YES        25
    2          400             NO        ---
    3          290             YES        10
    4          293             YES        7

Memory is Full, Remaining Processes cannot be accomodated
Total Internal Fragmentation is 42
Total External Fragmentation is 100
```

## MULTI-PROGRAMMING WITH VARIABLE PARTITIONING(MVT):

Multi-programming with variable partitioning is a continguous memory management technique in which the main memory is not divided into partitions and the processs is allocated a chunk of free memory that is big enough for it to fit.The space which is left is considered as the free space which can be further used by other processes.It is also provide the concept of compaction.In compaction the spaces that are free and the spaces which not allocated to the process are combined and single large memory space is made.

**CODE:**

```
#include<stdio.h>
#include<conio.h>
int main()
{
int ma,mp[10],i,temp,n=0;
char ch='y';
printf("\nEnter the total memory available (in Bytes):");
scanf("%d",&ma);
temp=ma;
for(i=0;ch=='y';i++,n++)
{
printf("\nEnter memory required for process %d (in Bytes):",i+1);
scanf("%d",&mp[i]);
if(mp[i]<=temp)
{
printf("\nMemory is allocated for Process %d:",i+1);
temp=temp-mp[i];
}
else
{
printf("\nMemory is Full");
break;
}
printf("\nDo you want to continue(y/n):");
scanf(" %c",&ch);
}
printf("\n\nTotal Memory Available -- %d",ma);
printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED ");
for(i=0;i<n;i++)
printf("\n \t%d\t\t%d",i+1,mp[i]);
printf("\n\nTotal Memory Allocated is %d",ma-temp);
printf("\nTotal External Fragmentation is %d",temp);
 getch();
}
```

**OUTPUT:**

```
Enter the total memory available (in Bytes):1000
Enter memory required for process 1 (in Bytes):400
Memory is allocated for Process 1:
Do you want to continue(y/n):y
Enter memory required for process 2 (in Bytes):275
Memory is allocated for Process 2:
Do you want to continue(y/n):y
Enter memory required for process 3 (in Bytes):550
Memory is Full
Total Memory Available -- 1000
  PROCESS          MEMORY ALLOCATED
       1                      400
       2                      275
Total Memory Allocated is 675
Total External Fragmentation is 325
```

| SNO | TASK | MARKS |
|---|---|---|
| 1 | Observation, Program writing and Execution | /10 |
| 2 | Record Submission | /5 |
| 3 | Viva-voce | /5 |
| 4 | Total | /20 |

# WEEK-12

**AIM:** Implement the following Contiguous Memory Allocation techniques a) Worst-fit b) Best-fit c) First fit

**WORST-FIT MEMORY ALLOCATION:**

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

**CODE:**

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
   int frag[max],b[max],f[max],i,j,nb,nf,temp;
   static int bf[max],ff[max];
   printf("Memory Management Scheme - Worst Fit\n");
   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:\tFRAGMENT");
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();
}
```

**OUTPUT:**

```
Memory Management Scheme - Worst Fit
Enter the number of blocks:3
Enter the number of files:2
Enter the size of the blocks:
Block 1:5
Block 2:2
Block 3:7
Enter the size of the files:
File 1:1
File 2:4
FILE-NO  FILE-SIZE   BLOCK-NO  BLOCK-SIZE    FRAGMENT
   1         1           1          5             4
   2         4           3          7             3
```

## BEST-FIT MEMORY ALLOCATION:

Best-Fit memory allocation method keeps the list in order by size -smallest to largest.In this method ,the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory ,making it able to use memory efficiently.Here the jobs are smallest to largest job.

## CODE:

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
static int bf[max],ff[max];
printf("Memory Management Scheme - Best Fit\n");
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\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();
}
```

**OUTPUT:**

```
Memory Management Scheme - Best Fit
Enter the number of blocks:3
Enter the number of files:2
Enter the size of the blocks:
Block 1:5
Block 2:2
Block 3:7
Enter the size of the files:
File 1:1
File 2:4
FILE NO  FILE SIZE   BLOCK NO   BLOCK SIZE    FRAGMENT
   1         1          2            2            1
   2         4          1            5            1
```

**FIRST-FIT MEMORY ALLOCATION:**

First-fit memory allocation method keeps the free/busy list of jobs organised by memory location,low-ordered to high-ordered memory.In this method,first job claims the first available memory with space more than or equal to it`s size.The operating system doesn`t search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

**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("Memory Management Scheme - First Fit\n");
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(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}
printf("\nFILE 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();
}
```

**OUTPUT:**

```
Memory Management Scheme - First Fit
Enter the number of blocks:3
Enter the number of files:2
Enter the size of the blocks:
Block 1:5
Block 2:2
Block 3:7
Enter the size of the files :
File 1:1
File 2:4
FILE NO  FILE SIZE   BLOCK NO    BLOCK SIZE    FRAGMENT
   1         1           3            7             6
   2         4           1            5             1
```

| SNO | TASK | MARKS |
|---|---|---|
| 1 | Observation, Program writing and Execution | /10 |
| 2 | Record Submission | /5 |
| 3 | Viva-voce | /5 |
| 4 | Total | /20 |