Program 1:

a. Write a program to read data from the standard input device and write it on the screen (using read()/write() system calls)

write()/read() system call

read() and write() system calls are used to read and write data respectively to a file descriptor. To understand the concept of write ()/read() system calls let us first start with write() system call.

write()

write() system call is used to write to a file descriptor. In other words **write()** can be used to write to any file (all hardware are also referred as file in Linux) in the system but rather than specifying the file name, you need to specify its file descriptor.

Syntax:

```
#include<unistd.h>
ssize_t write (int fd, const void *buf,size_t count);
```

The first parameter (fd) is the file descriptor where the file is to be written. The data that is to be written is specified in the second parameter. Finally, the third parameter is the total bytes that are to be written.

read()

The use of **read()** system call is to read from a file descriptor. The working is same as write(), the only difference is **read()** will read the data from file pointed to by file descriptor.

Syntax:

```
#include<unistd.h>
ssize_t read(int fd,const void  *buf,size_t count);
```

The first parameter is the file descriptor. The second parameter is the buffer where the read data will be saved. Lastly, the third parameter is the number of bytes that is to be read.

Program:

```
#include<unistd.h>
        int main()
        {
        int nread;
        char buff[20];
        nread=read(0,buff,10);//read 10 bytes from standard input device(keyboard) and store it
                                in buffer(buff)
        write (1,buff,nread);//print 10 bytes from the buffer on the screen
        }
```

b. Write a program to print 10 characters starting from the 10th character from a file(lseek() system call)

lseek() system call repositions the read/write file offset i.e., it changes the positions of the read/write pointer within the file. In every file any read or write operations happen at the position pointed to by the pointer. lseek () system call helps us to manage the position of this pointer within a file. e.g., let's suppose the content of a file F1 is "1234567890" but you want the content to be "12345hello". You simply can't open the file and write "hello" because if you do so then "hello" will be written in the very beginning of the file. This means you need to reposition the pointer after '5' and then start writing "hello". lseek () will help to reposition the pointer and write () will be used to write "hello"

Syntax:

```
        #include<sys/types.h>
        #include<unistd.h>
        off_t lseek (int fd,off_t offset,int whence);
```

The first parameter is the file descriptor of the file, which you can get using open () system call.

The second parameter specifies how much the pointer is to be moved.

The third parameter is the reference point of the movement i.e., beginning of file(SEEK_SET),current position(SEEK_CUR) of pointer or end of file(SEEK_END).

Examples:

- lseek(fd,5,SEEK_SET) – this moves the pointer 5 positions ahead starting from the beginning of the file
- lseek(fd,5,SEEK_CUR) – this moves the pointer 5 positions ahead from the current position in the file
- lseek(fd,-5,SEEK_CUR) – this moves the pointer 5 positions back from the current position in the file
- lseek(fd,-5,SEEK_END) -> this moves the pointer 5 positions back from the end of the file
- On success, lseek() returns the position of the pointer within the file as measured in bytes from the beginning of the file. But, on failure, it returns -1.

Program:

```
//Let the contents of the file F1 be "1234567890abcdefghijxxxxxxxx". This means we want
the output to be "abcdefghij".
//Note: the first character '1' is at 0th position

#include<unistd.h>
#include<fcntl.h>
#include<sys/types.h
#include<sys/stat.h>
#include<stdio.h>
int main()
{
int n,f,f1;
char buff[10];
f=open("seeking",O_RDWR);
f1=lseek(f,10,SEEK_SET);
printf("Pointer is at %d position\n",f1);
read(f,buff,10);
write(1,buff,10);
}
```

Program 2:

Write a program to implement IPC using shared memory.

Inter Process Communication through shared memory is a concept where two or more process can access the common memory and communication is done via this shared memory where changes made by one process can be viewed by another process.
The problem with pipes, fifo and message queue – is that for two process to exchange information, the information has to go through the kernel.

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

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

SYSTEM CALLS USED ARE:

ftok(): is use to generate a unique key.

shmget(): int shmget(key_t,size_tsize,intshmflg);
upon successful completion, shmget() returns an identifier for the shared memory segment.

shmat(): Before you can use a shared memory segment, you have to attach yourself to it using shmat().
void *shmat(int shmid ,void *shmaddr ,int shmflg);
shmid is shared memory id.
shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

shmdt(): When you're done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void *shmaddr);

shmctl(): when you detach from shared memory, it is not destroyed. So, to destroy shmctl() is used.

```
shmctl(int shmid,IPC_RMID,NULL);
```

Shared Memory for Writer Process

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>

int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
        //creates shared memory segment with key 2345, having size 1024 bytes. IPC_CREAT
is used to create the shared segment if it does not exist. 0666 are the permisions on the
shared segment.
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0);  //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
                //this prints the address where th segment is attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

How it works?

shmget() function creates a segment with key 2345, size 1024 bytes and read and write permissions for all users. It returns the identifier of the segment which gets store in shmid. This identifier is used in shmat() to attach the shared segment to the address space of the process. NULL in shmat() means that the OS will itself attach the shared segment at a suitable address of this process. Then some data is read from the user using read() system call and it is finally written to the shared segment using strcpy() function.

Output:

```
Key of shared memory is 0
Process attached at x7ffe04fb000

Enter some data to write to shared memory
Hello World
You wrote: Hello World
```

Shared Memory for Reader Process

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

How it works?

shmget () here generates the identifier of the same segment as created in writer process. The same key value must be given. The only change is, do not write IPC_CREAT as the shared memory segment is already created. Next, shmat () attaches the shared segment to the current process. After that, the data is printed from the shared segment. The output, we obtain is the same data that is written while executing the Writer process.

Output

```
Key of Shared memory is 0
Process attached at 0x7f76b4292999
Data read from shared memory is: Hello World
```

Program 3:

Implement the Producer & consumer Problem (Semaphore)

Description:

Producer consumer problem is a synchronization problem. There is a fixed size buffer where producer produces items and that is consumed by the 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.

```
#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 \t3. 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;
                 }
            }
    }
```

Program 4:

Implement the solution to dining philosopher's problem using monitors.

Description:

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 presentation 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 centre 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 neighbours). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbour. 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.

ALGORITHM:

```
Step 1: Start the program.
Step 2: Define the number of philosophers.
Step 3: Declare one thread per philosopher.
Step 4: Declare one chop sticks per philosopher.
Step 5: When a philosopher is hungry.
        i. See if chopsticks on both sides are free.
        ii. Acquire both chopsticks or.
        iii. Eat.
        iv. restore the chopsticks.
        v. If chopsticks aren't free.
Step 6: Wait till they are available.
```

Step 7: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int one();
int two() ;

int tph, philname[20], status[20], howhung, hu[20], cho;
int 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("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your
choice:");
            scanf("%d", &cho);
            switch(cho)
            {
               case 1:
                     one();
                     break;
               case 2:
                      two();
                      break;
               case 3:
                  exit(0);
               default:
                  printf("\nInvalid option..");
            }
         }
         while(1);
         }

}

int 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]]);
```

```
    }
}



int 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]]);
                }

        }
    }
    }
    }
```

Program 5:

Implement the various CPU Scheduling Algorithms (FCFS, RR)

FCFS:

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

Program 6:

Round Robin:

```
#include<stdio.h>
main()
{
int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
float awt=0,att=0,temp=0;
clrscr();
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
        {
        printf("\nEnter Burst Time for process %d -- ", i+1);
        scanf("%d",&bu[i]);
        ct[i]=bu[i];
        }
 printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
         if(max<bu[i])
                max=bu[i];
for(j=0;j<(max/t)+1;j++)
        for(i=0;i<n;i++)
                if(bu[i]!=0)
                         if(bu[i]<=t)
                                {
                                 tat[i]=temp+bu[i];
                                 temp=temp+bu[i];
                                bu[i]=0;
                                }
                        else
```

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

Program 7:

Implement Bankers Algorithm for Deadlock Avoidance

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.

```
#include<stdio.h>
struct file
 {
 int all[10];
 int max[10];
 int need[10];
 int flag;
};
 void main()
{
struct file f[10];
int fl;
 int i, j, k, p, b, n, r, g, cnt=0, id, newr;
int avail[10],seq[10];
```

```
clrscr();
printf("Enter number of processes -- ");
 scanf("%d",&n);
printf("Enter number of resources -- ");
scanf("%d",&r);
for(i=0;i<n;i++)
{
printf("Enter details for P%d",i);
 printf("\nEnter allocation\t -- \t");
for(j=0;j<r;j++)
scanf("%d",&f[i].all[j]);
printf("Enter Max\t\t -- \t");
for(j=0;j<r;j++)
scanf("%d",&f[i].max[j]);
 f[i].flag=0;
 }
printf("\nEnter Available Resources\t -- \t");
 for(i=0;i<r;i++)
 scanf("%d",&avail[i]);
printf("\nEnter New Request Details -- ");
printf("\nEnter pid \t -- \t");
scanf("%d",&id);
printf("Enter Request for Resources \t -- \t");
for(i=0;i<r;i++)
{
scanf("%d",&newr);
f[id].all[i] += newr;
 avail[i]=avail[i] - newr;
 }
for(i=0;i<n;i++)
 {
 for(j=0;j<r;j++)
{
 f[i].need[j]=f[i].max[j]-f[i].all[j];
if(f[i].need[j]<0)
 f[i].need[j]=0;
 }
 }
cnt=0;
 fl=0;
 while(cnt!=n)
{
g=0;
for(j=0;j<n;j++)
 {
if(f[j].flag==0)
 {
b=0;
for(p=0;p<r;p++)
 {
if(avail[p]>=f[j].need[p])
 b=b+1; else b=b-1;
 }
if(b==r)
 {
 printf("\nP%d is visited",j);
 seq[fl++]=j;
 f[j].flag=1;
for(k=0;k<r;k++)
avail[k]=avail[k]+f[j].all[k];
 cnt=cnt+1;
printf("(");
for(k=0;k<r;k++)
printf("%3d",avail[k]);
printf(")"); g=1;
 }
 }
}
```

```
 if(g==0)
{
 printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
 printf("\n SYSTEM IS IN UNSAFE STATE");
 goto y;
}
 }
printf("\nSYSTEM IS IN SAFE STATE");
printf("\nThe Safe Sequence is -- (");
for(i=0;i<fl;i++)
 printf("P%d ",seq[i]); printf(")");
 y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
for(i=0;i<n;i++)
{
 printf("P%d\t",i);
 for(j=0;j<r;j++)
printf("%6d",f[i].all[j]);
for(j=0;j<r;j++)
printf("%6d",f[i].max[j]);
 for(j=0;j<r;j++)
printf("%6d",f[i].need[j]); printf("\n"); }
 getch();
}
```

INPUT

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

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

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

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

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

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

OUTPUT

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

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

Program 8:

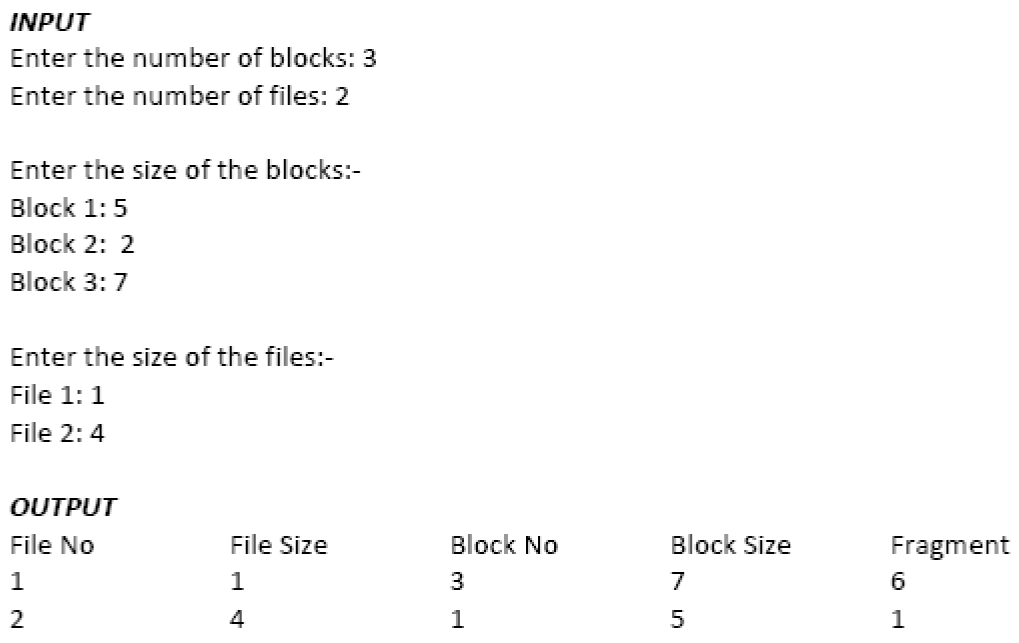
Implement the following Memory Allocation Methods for fixed partition

a) First Fit b) Worst Fit

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.

a) <u>Worst Fit</u>

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
static int bf[max],ff[max];
clrscr();
 printf("\n\tMemory Management Scheme - Worst Fit");
printf("\nEnter the number of blocks:");
 scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
        {
         printf("Block %d:",i);
         scanf("%d",&b[i]);
        }
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
        {
        printf("File %d:",i);
        scanf("%d",&f[i]);
         }
for(i=1;i<=nf;i++)
        {
        for(j=1;j<=nb;j++)
                    {
                    if(bf[j]!=1) //if bf[j] is not allocated
                                {
                                temp=b[j]-f[i];
                                if(temp>=0)
                                            if(highest<temp)
                                                    {
                                                    ff[i]=j;
                                                    highest=temp;
                                                    }
                                }
                    }
        frag[i]=highest;
        bf[ff[i]]=1;
        highest=0;
        }
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
 for(i=1;i<=nf;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
 }
```



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

Enter the size of the blocks:-
Block 1: 5
Block 2:  2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4

OUTPUT
File No        File Size      Block No       Block Size     Fragment
1              1              3              7              6
2              4              1              5              1
```

b) **<u>First Fit</u>**

c) **Program 9:**

```
#include<stdio.h>
#include<conio.h>
 #define max 25
 void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
static int bf[max],ff[max];
 clrscr();
printf("\n\tMemory Management Scheme - First Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
        {
         printf("Block %d:",i); scanf("%d",&b[i]);
        }
 printf("Enter the size of the files :-\n");
         for(i=1;i<=nf;i++)
        {
        printf("File %d:",i);
         scanf("%d",&f[i]);
        }
 for(i=1;i<=nf;i++)
         {
        for(j=1;j<=nb;j++)
        {
         if(bf[j]!=1)
                    {
                     temp=b[j]-f[i];
                     if(temp>=0)
                    {
                    ff[i]=j;
                     break;
                     }
         }
 }
frag[i]=temp;
 bf[ff[i]]=1;
 }
 printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
 printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
 }
```

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

Enter the size of the blocks:-
Block 1: 5
Block 2:  2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4

OUTPUT
File No         File Size       Block No        Block Size      Fragment
1               1               1               5               4
2               4               3               7               3
```

Program 10:

Implement the following Page Replacement Algorithms

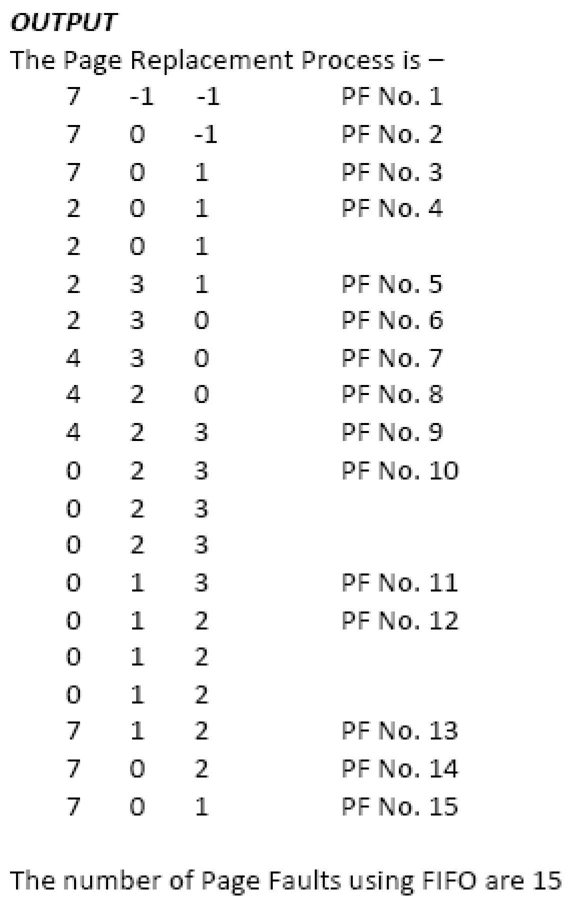
a) FIFO   b) LRU

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.

a) FIFO

```
#include<stdio.h>
 #include<conio.h>
main()
{
 int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
clrscr();
 printf("\n Enter the length of reference string -- ");
scanf("%d",&n); printf("\n Enter the reference string -- ");
for(i=0;i<n;i++)
scanf("%d",&rs[i]);
printf("\n Enter no. of frames -- ");
scanf("%d",&f);
 for(i=0;i<f;i++)
m[i]=-1;
printf("\n The Page Replacement Process is -- \n");
for(i=0;i<n;i++)
{
for(k=0;k<f;k++)
{
 if(m[k]==rs[i]) break;
 }
 if(k==f)
{
m[count++]=rs[i];
 pf++;
}
 for(j=0;j<f;j++)
 printf("\t%d",m[j]);
 if(k==f)
printf("\tPF No. %d",pf); printf("\n");
 if(count==f)
count=0;
}
printf("\n The number of Page Faults using FIFO are %d",pf);
 getch();
}
```



OUTPUT

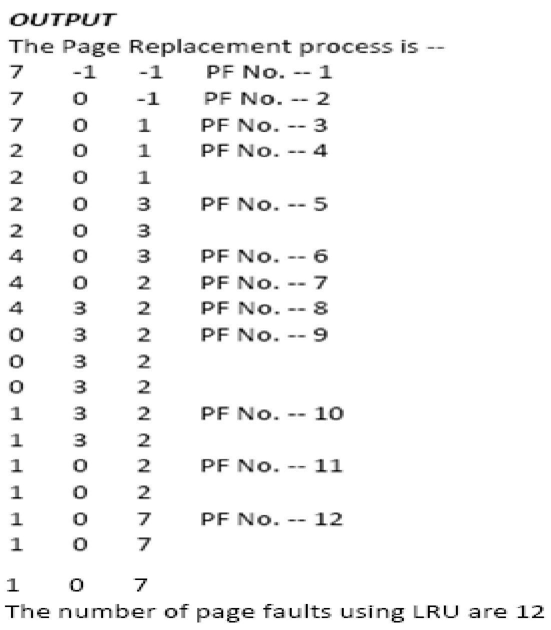
```
The Page Replacement Process is –
      7    -1    -1            PF No. 1
      7     0    -1            PF No. 2
      7     0     1            PF No. 3
      2     0     1            PF No. 4
      2     0     1
      2     3     1            PF No. 5
      2     3     0            PF No. 6
  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

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

b) LRU

c) Program 11:

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

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



```
OUTPUT
The Page Replacement process is --
7     -1    -1    PF No. -- 1
7     0     -1    PF No. -- 2
7     0     1     PF No. -- 3
2     0     1     PF No. -- 4
2     0     1
2     0     3     PF No. -- 5
2     0     3
4     0     3     PF No. -- 6
4     0     2     PF No. -- 7
4     3     2     PF No. -- 8
0     3     2     PF No. -- 9
0     3     2
0     3     2
1     3     2     PF No. -- 10
1     3     2
1     0     2     PF No. -- 11
1     0     2
1     0     7     PF No. -- 12
1     0     7

1     0     7
The number of page faults using LRU are 12
```

Program 12

Implement the following Disk Scheduling Algorithms:

a) SSTF Scheduling b) SCAN Scheduling

SSTF Scheduling

SSTF stands for Shortest Time First which very uses full of learning about how the disk drive manages the data having the shortest seek time.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested. ‘head’ is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int main()
{
int queue[100],t[100],head,seek=0,n,i,j,temp;
float avg;
// clrscr();
printf("*** SSTF Disk Scheduling Algorithm ***\n");
printf("Enter the size of Queue\t");
scanf("%d",&n);
printf("Enter the Queue\t");
for(i=0;i<n;i++)
{
scanf("%d",&queue[i]);
}
printf("Enter the initial head position\t");
scanf("%d",&head);
for(i=1;i<n;i++)
t[i]=abs(head-queue[i]);
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
if(t[i]>t[j])
{
temp=t[i];
t[i]=t[j];
t[j]=temp;
temp=queue[i];
queue[i]=queue[j];
queue[j]=temp;
}
}
}
for(i=1;i<n-1;i++)
{
seek=seek+abs(head-queue[i]);
head=queue[i];
}
printf("\nTotal Seek Time is%d\t",seek);
avg=seek/(float)n;
printf("\nAverage Seek Time is %f\t",avg);
return 0;
}
```

OUTPUT:

```
*** SSTF Disk Scheduling Algorithm ***
Enter the size of Queue 5
Enter the Queue 10 17 2 15 4
Enter the initial head position 3
Total Seek Time is14
Average Seek Time is 2.800000
```

RESULT:

Thus the program was executed and verified successfully.

Program 13:

SCAN Scheduling

SCAN DISK SCHEDULING ALGORITHM

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

INPUT

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

OUTPUT:

Tracks Traversed	**Difference Between tracks**
150	50

160	10
184	24
90	94
70	20
60	10
58	2
55	3
18	37

Program 14:

Implement the following File Allocation Strategies

a) Sequential b) Indexed

OBJECTIVE Write a C program to simulate the following file allocation strategies.

a) Sequential b) Indexed

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

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

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

SEQUENTIAL FILE ALLOCATION

```
 #include<stdio.h>
 #include<conio.h>
 struct fileTable
 {
char name[20];
int sb, nob;
 }ft[30];
void main()
{
int i, j, n;
char s[20];
clrscr();
printf("Enter no of files :");
scanf("%d",&n);
for(i=0;i<n;i++)
 {
printf("\nEnter file name %d :",i+1);
scanf("%s",ft[i].name);
printf("Enter starting block of file %d :",i+1);
scanf("%d",&ft[i].sb);
printf("Enter no of blocks in file %d :",i+1);
scanf("%d",&ft[i].nob);
}
printf("\nEnter the file name to be searched -- ");
scanf("%s",s);
for(i=0;i<n;i++)
if(strcmp(s, ft[i].name)==0)
break;
if(i==n)
printf("\nFile Not Found");
else
{
printf("\nFILE NAME START BLOCK NO OF BLOCKS BLOCKS OCCUPIED\n");
printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob);
for(j=0;j<ft[i].nob;j++)
 printf("%d, ",ft[i].sb+j);
 }
getch();
 }
```

INPUT: Enter no of files :3

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

Enter file name 2 :B
Enter starting block of file 2 :102
 Enter no of blocks in file 2 :4
Enter file name 3 :C
Enter starting block of file 3 :60
Enter no of blocks in file 3 :4
Enter the file name to be searched -- B
```

OUTPUT:

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

Program 15

INDEXED FILE ALLOCATION

```
 #include<stdio.h>
#include<conio.h>
struct fileTable
{
char name[20];
 int nob, blocks[30];
}ft[30];
void main()
```

```
{
int i, j, n;
char s[20];
 clrscr();
printf("Enter no of files :");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
printf("\nEnter file name %d :",i+1);
scanf("%s",ft[i].name);
 printf("Enter no of blocks in file %d :",i+1);
scanf("%d",&ft[i].nob);
printf("Enter the blocks of the file :");
for(j=0;j<ft[i].nob;j++)
scanf("%d",&ft[i].blocks[j]);
 }
printf("\nEnter the file name to be searched -- ");
scanf("%s",s);
for(i=0;i<n;i++)
if(strcmp(s, ft[i].name)==0)
break;
if(i==n)
printf("\nFile Not Found");
else
{
printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED");
printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob);
for(j=0;j<ft[i].nob;j++)
 printf("%d, ",ft[i].blocks[j]);
}
getch();
 }
```

INPUT:

```
Enter no of files : 2
Enter file 1 : A
Enter no of blocks in file 1 : 4
Enter the blocks of the file 1 : 12 23 9 4

Enter file 2 : G
Enter no of blocks in file 2: 5
Enter the blocks of the file 2: 88 77 66 55 44
 Enter the file to be searched: G
```

OUTPUT:

```
FILE NAME      NO OF BLOCKS                  BLOCKS OCCUPIED
    G               5                        88, 77, 66, 55, 44 13
```