OS LAB PROGRAMS

1)Demonstrate the process creation and termination using system calls- fork(), vfork(), getpid(), waitpid(), exec, exit(), return 0

a)

```
#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

int main(){

 pid_t pid;

 int status;


 pid =fork();

 if(pid<0){

      printf("Error :fork() failed.\n");

      return 1;    }

    else if(pid==0){

                          printf("This is the child process with PID:%d\n",getpid());

                          printf("Parent process PID:%d\n",getpid());

                           execlp("/bin/ls","ls",NULL);

                           printf("This should not be printed if exec() is successful.\n");

            return 0;   }

     else{

                           printf("This is the parent process with PID:%d\n",getpid());

                           printf("Child process PID:%d\n",pid);

                           wait(&status);

                           printf("Child process exited the status:%d\n",status);

                           return 0;

     }

}
```

b) vfork()

```
#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<sys/types.h>

int main(){

 pid_t pid;

 pid =vfork();

 if(pid==-1){

    perror("vfork");

    return 1;

   }

   else if(pid==0){

                printf("Child process :Hello,I'm the child!\n");

                printf("Child process :My PID is %d\n",getpid());

                printf("Child process :My parent's PID is %d\n",getppid());

        _exit(0);

    }

    else{


                printf("Parent process :Hello,I'm the Parent!\n");

                printf("Parent process :My PID is %d\n",getpid());

                printf("Parent process :My child's PID is %d\n",pid);

        int status;

        waitpid(pid,&status,0);

        if(WIFEXITED(status)){

                printf("Parent process :Child process terminated normally.\n");

                }

        else{
```

```
                    printf("Parent process :Child process terminated abnormally.\n");
                    }
      }
      return 0;
}
```

2) Write a C program to stimulate Inter-Process Communication(IPC) techniques: Pipes, Messages Queues and Shared Memory

write.c

```
#include<fcntl.h>

#include<unistd.h>

#include<sys/stat.h>

#include<sys/types.h>



int main(){


  int fd;

  char *myfifo="/tmp/myfifo";

  mkfifo(myfifo,0666);

  fd=open(myfifo,O_WRONLY);

  write(fd,"Hello", sizeof('Hello'));

  close(fd);

  unlink(myfifo);

  return 0;


}
```

read.c

```
#include<fcntl.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<stdio.h>
#define MAX_BUF 1024
int main(){

  int fd;
  char *myfifo="/tmp/myfifo";
  char buf[MAX_BUF];
  fd=open(myfifo,O_RDONLY);
  read(fd,buf,MAX_BUF);
  printf("Received:%s\n",buf);
  close(fd);
  return 0;
}
```

3)Stimulate the following CPU scheduling algorithms 1.FCFS 2.SJF 3.Priority 4.RoundRobin.

Calculate the avg. waiting time, avg. Turn around time, avg.Response time for each algorithm.

```
#include<stdio.h>
void fcfs(int processes[],int n,int burst_time[]){
 int waiting_time[n],turnaround_time[n],total_waiting_time=0,total_turnaround_time=0;
 waiting_time[0]=0;
 for(int i=1;i<n;i++){
        waiting_time[i]=burst_time[i-1]+ waiting_time[i-1];
        total_waiting_time= total_waiting_time+waiting_time[i];
        }
 for(int i=0;i<n;i++){
        turnaround_time[i] = burst_time[i] + waiting_time[i];
         total_turnaround_time += turnaround_time[i];
        }
        printf("First Come,First Served(FCFS) scheduling Algorithm\n");
        printf("--------------------------------------------------\n");
        printf("Process\t Burst Time\t Waiting Time\t Turnaround Time\n");

        for(int i=0;i<n;i++)
        {
                printf("%d\t %d\t\t %d\t\t%d \n",processes[i],burst_time[i] ,
waiting_time[i],turnaround_time[i]);
        }

        printf("Average Waiting Time : %.2f\n",(float)total_waiting_time/n);
        printf("Average Turnaround Time : %.2f\n",(float)total_turnaround_time/n);
        printf("/n");
}
```

```
void sjf(int processes[],int n,int burst_time[]){
 int waiting_time[n],turnaround_time[n],completion_time[n],total_waiting_time=0,total_turnaround_time=0;
 for(int i=0;i<n;i++){
        int shortest_job_index=i;

 for(int j=i+1;j<n;j++){
        if(burst_time[j]<burst_time[shortest_job_index])
                shortest_job_index=j;
        }
 int temp= burst_time[i] ;
 burst_time[i]   =burst_time[shortest_job_index] ;
 burst_time[shortest_job_index] =temp;
 temp=processes[i] ;
 processes[i]=processes[shortest_job_index];
 processes[shortest_job_index]=temp;
 }
 waiting_time[0]=0;
  for(int i=1;i<n;i++){
        waiting_time[i]=burst_time[i-1]+ waiting_time[i-1];
        total_waiting_time= total_waiting_time+waiting_time[i];
        }
 for(int i=0;i<n;i++){
        turnaround_time[i] = burst_time[i] + waiting_time[i];
        total_turnaround_time = total_turnaround_time + turnaround_time[i];
        }
```

```
printf("Shortest Job First(SJF) scheduling Algorithm\n");

printf("---------------------------------------------------\n");

printf("Process\t Burst Time\t Waiting Time\t Turnaround Time\n");


 for(int i=0;i<n;i++){

                printf("%d\t %d\t\t  %d\t\t%d \n",processes[i],burst_time[i] ,
waiting_time[i],turnaround_time[i]);

        }


        printf("Average Waiting Time : %.2f\n",(float)total_waiting_time/n);

        printf("Average Turnaround Time : %.2f\n",(float)total_turnaround_time/n);

        printf("\n");

        }



void roundRobin(int processes[],int n,int burst_time[],int quantum){

 int
remaining_time[n],waiting_time[n],turnaround_time[n],total_waiting_time=0,total_turnaround_tim
e=0;


 for(int i=0;i<n;i++){

 remaining_time[i]=burst_time[i];

 }

 int time=0;


        while(1){

        int all_processes_completed=1;


 for(int i=0;i<n;i++){

        if(remaining_time[i]>0){
                all_processes_completed=0;
                        if(remaining_time[i]>quantum)
                        {
                                time += quantum;
                                 remaining_time[i]-=quantum;
                         }
                         else{
                                 time +=remaining_time[i];
                                 waiting_time[i]=time-burst_time[i];
                                 remaining_time[i]=0;
                         }
                }
         }
        if(all_processes_completed)
        {
                break;
        }
 }

 for(int i=0;i<n;i++){
        turnaround_time[i] = burst_time[i] + waiting_time[i];
        total_waiting_time+= waiting_time[i];
        total_turnaround_time += turnaround_time[i];
        }
printf("Round Robin scheduling Algorithm\n");
printf("----------------------------------------------------\n");
printf("Process\t Burst Time\t Waiting Time\t Turnaround Time\n");

  for(int i=0;i<n;i++){
    printf("%d\t %d\t\t  %d\t\t%d T\n",processes[i],burst_time[i] ,
waiting_time[i],turnaround_time[i]);
    }


printf("Average Waiting Time : %.2f\n",(float)total_waiting_time/n);
printf("Average Turnaround Time : %.2f\n",(float)total_turnaround_time/n);
printf("/n");
}



int main(){
     int n;
    printf("Enter the no.of processes:");
    scanf("%d",&n);
    int processes[n],burst_time[n];
     printf("Enter the burst_time for each process:\n");
     for(int i=0;i<n;i++){
            printf("Process %d: ",i+1);
            scanf("%d",&burst_time[i]);
             processes[i]=i+1;
    }

    int quantum;
    printf("Enter the quantum time for Round Robin:\n");
    scanf("%d",&quantum);
    printf("\n");
    fcfs(processes,n,burst_time);
    sjf(processes,n,burst_time);
    roundRobin(processes,n,burst_time,quantum);
```

```
return 0;

}
```

3)Priority algorithm

```
#include<stdio.h>
int main()
{
    int p[20],bt[20],pri[20],wt[20],tat[20],i,k,n,temp;
    float wtavg,tatavg;
    printf("Enter the no.of processes---");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        p[i]=i;
        printf("Enter the Burst Time & Priority of process %d---",i);
        scanf("%d%d",&bt[i],&pri[i]);
    }
    for(i=0;i<n;i++)
        for(k=i+1;k<n;k++)
            if(pri[i]>pri[k])
            {
                temp=p[i];
                p[i]=p[k];
                p[k]=temp;
                temp=bt[i];
                bt[i]=bt[k];
                bt[k]=temp;
                temp=pri[i];
                pri[i]=pri[k];
                pri[k]=temp;
            }
        wtavg=wt[0]=0;
        tatavg=tat[0]=bt[0];
        for(i=1;i<n;i++)
        {
                wt[i]=wt[i-1]+bt[i-1];
                tat[i]=tat[i-1]+bt[i];
                wtavg=wtavg+wt[i];
                tatavg=tatavg+tat[i];
        }
        printf("\nPROCESS\t\tPRIORITY\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("\nAverage waiting Time is----------%f",wtavg/n);
                printf("\nAverage Turnaround Time  is %f",tatavg/n);
                return 0;
}
```

4) Write a C program to implement basic UNIX system calls- read(), write(), open(), close()

```
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
#include<unistd.h>

int main(){
    int fd,ret;
    char buffer[20];
    fd=creat("example.txt",0644);
    if(fd==-1){
        perror("creat");
        exit(EXIT_FAILURE);
    }
    ret=close(fd);
    if(ret==-1){
        perror("close");
        exit(EXIT_FAILURE);
    }

    fd=open("example.txt",O_RDWR);
    if(fd==-1){
        perror("open");
        exit(EXIT_FAILURE);
    }

    ret=lseek(fd,0,SEEK_SET);
    if(ret==-1){
        perror("lseek");
        exit(EXIT_FAILURE);
    }

    ret=read(fd,buffer,13);
    if(ret==-1){
        perror("read");
        exit(EXIT_FAILURE);
    }
    buffer[ret]='\0';

    printf("Read from file :%s\n",buffer);

    ret=close(fd);
    if(ret==-1){
        perror("close");
        exit(EXIT_FAILURE);
    }
    return 0;

}
```

5)Write a C program to implement UNIX directory API’s – opendir, closedir, readdir, mkdir

```
#include<stdio.h>
#include<stdlib.h>
#include<dirent.h>
#include<sys/stat.h>
#include<errno.h>

void listDirectory(const char *path){
	DIR *dir = opendir(path);
	if(dir==NULL){
		perror("opendir");
		return;
	}

	struct dirent *entry;
	while((entry = readdir(dir))!=NULL){
		printf("%s\n",entry->d_name);
		}
	if(closedir(dir)==-1){
		perror("closedir");
	}
}

void createDirectory(const char *path)
{
	if(mkdir(path,0755)==-1){
		if(errno==EEXIST){
			printf("Directory %s already exists,\n",path);
```

```
        }
        else{
            perror("mkdir");
            }
        }
        else{
        printf("Directory %s created succesfully:\n",path);
    }
}

int main(){
    const char *dirPath = "./testdir";

    createDirectory(dirPath);
    printf("Listing current directory contents:\n");
    listDirectory(".");

    printf("\nListing new Directory contents:\n");
    listDirectory(dirPath);

    return 0;
}
```

6)Demonstrate the following Classical problems of synchronization using semaphores.

a. Producer-Consumer b. Dining Philosopher

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2
#define NUM_PHILOSOPHERS 5

int buffer[BUFFER_SIZE];
int in =0;
int out = 0;

sem_t emptySlots;
sem_t filledSlots;
sem_t bufferMutex;
sem_t mutex;
sem_t rwMutex;

void *producer(void *producerId )
{
	int id = *(int *)producerId;
	int item=0;

	while(1){
```

```
            item++;
            sem_wait(&emptySlots);
            sem_wait(&bufferMutex);

            buffer[in]=item;
            printf("Producer %d produced item %d\n",id,item);
            in = (in+1)%BUFFER_SIZE;
            sem_post(&bufferMutex);
            sem_post(&filledSlots);
            usleep(rand() % 1000000);
            }
            }
void *consumer(void *consumerId){
int id=*(int *)consumerId;
int item;
while(1){
           sem_wait(&filledSlots);
            sem_wait(&bufferMutex);

            item=buffer[out];
            printf("Consumer %d consumed item %d\n",id,item);
            out=(out+1) %BUFFER_SIZE;
            sem_post(&bufferMutex);
            sem_post(&emptySlots);
            usleep(rand() % 1000000);
           }
}

enum {THINKING,HUNGRY,EATING}state[5];
```

```
sem_t philMutex;

sem_t philSem[5];

void test (int id)

{

    if(state[id] == HUNGRY && state[(id+1)%5]!=EATING && state[(id+4)%5]!=EATING)

    {

        state[id]=EATING;

        sem_post(&philSem[id]);

        }

}


void *philosopher(void *philosopherId)

{

    int id=*(int *)philosopherId;

    while(1)

    {

    printf("Philosopher %d is thinking\n" ,id);

    usleep(rand() % 1000000);

    sem_wait(&philMutex);

    state[id]=HUNGRY;

    printf("Philosopher %d is hungry\n",id);

    test(id);

    sem_post(&philMutex);

    sem_wait(&philSem[id]);

    printf("Philosopher %d is eating\n" ,id);

    usleep(rand() % 1000000);

    sem_post(&philSem[id]);

    printf("Philosopher %d finished eating\n" ,id);
    }
}

int main(){
    sem_init(&emptySlots,0,BUFFER_SIZE);
    sem_init(&filledSlots,0,0);
    sem_init(&bufferMutex,0,1);
    pthread_t producers[NUM_PRODUCERS];
    pthread_t consumers[NUM_CONSUMERS];
    int producerIds[NUM_PRODUCERS];
    int consumerIds[NUM_CONSUMERS];
    for(int i=0;i<NUM_PRODUCERS;i++){
        producerIds[i]=i+1;
        pthread_create(&producers[i],NULL,producer,(void *)&producerIds[i]);
    }
    for(int i=0;i<NUM_CONSUMERS;i++){
        consumerIds[i]=i+1;
        pthread_create(&consumers[i],NULL,consumer,(void *)&consumerIds[i]);
    }

    sem_init(&philMutex,0,1);
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopherIds[NUM_PHILOSOPHERS];
    for(int i=0;i<NUM_PHILOSOPHERS;i++)
    {
        philosopherIds[i]=i+1;
        sem_init(&philSem[i],0,0);

        pthread_create(&philosophers[i],NULL,philosopher,(void *)&philosopherIds[i]);
```

```
}
for(int i=0;i<NUM_PRODUCERS;i++)
{		pthread_join(producers[i],NULL);
		}

for(int i=0;i<NUM_CONSUMERS;i++)
{		pthread_join(consumers[i],NULL);
		}
 for(int i=0;i<NUM_PHILOSOPHERS;i++)
{
		pthread_join(philosophers[i],NULL);
}

sem_destroy(&emptySlots);
sem_destroy(&filledSlots);
sem_destroy(&bufferMutex);
sem_destroy(&mutex);
sem_destroy(&rwMutex);
sem_destroy(&philMutex);
 for(int i=0;i<NUM_PHILOSOPHERS;i++)
{
		sem_destroy(&philSem[i]);
}
return 0;
}
```

Demonstrate following page replacement algorithms:

a. FIFO, b. LRU, c. OPTIMAL.

a)FIFO

```
#include<stdio.h>
int fr[3];
int main()
{
    void display();
    int i,j,page[12]={2,3,2,1,5,2,4,5,3,2,5,2};
    int flag1=0,flag2=0,pf=0,frsize=3,top=0;
    for(i=0;i<3;i++)
    {
        fr[i]=-1;
    }
    for(j=0;j<12;j++)
    {
        flag1=0;
        flag2=0;
        for(i=0;i<3;i++)
        {
            if(fr[i]==page[j])
            {
                flag1=1;
                flag2=1;
                break;
            }
        }
        if(flag1==0)
        {
            for(i=0;i<frsize;i++)
            {
                if(fr[i]==-1)
                {
                    fr[i]=page[j];
                    flag2=1;
                    break;
                }
            }
        }
        if(flag2==0)
        {
            fr[top]=page[j];
            top++;
            pf++;
            if(top>=frsize)
            top=0;
        }
        display();
    }
    printf("\nNumber of page faults:%d\n",pf+frsize);
    return 0;
}
void display()
{
    int i;
    printf("\n");
```

```
        for(i=0;i<3;i++)

        printf("%d\t",fr[i]);

}
```

b)LRU

```
#include<stdio.h>
int fr[3];
int main()
{
    void display();
    int i,j,p[12]={2,3,2,1,5,2,4,5,3,2,5,2},fs[3];
    int flag1=0,flag2=0,pf=0,frsize=3,index,k,l;
    for(i=0;i<3;i++)
    {
        fr[i]=-1;
    }
    for(j=0;j<12;j++)
    {
        flag1=0,flag2=0;
        for(i=0;i<3;i++)
        {
            if(fr[i]==p[j])
            {
                flag1=1;
                flag2=1;
                printf("\tflag %d-%d",flag1,flag2);
                break;
            }
        }
        if(flag1==0)
        {
            for(i=0;i<3;i++)
            {
```

```
            if(fr[i]==-1)
            {
                fr[i]=p[j];
                flag2=1;
                break;
            }
        }
    }
    if(flag2==0)
    {
        for(i=0;i<3;i++)
        fs[i]=0;
        for(k=j-1,l=1;l<=frsize-1;l++,k--)
        {
            for(i=0;i<3;i++)
            {
                if(fr[i]==p[k])
                    fs[i]=1;
            }
        }
        for(i=0;i<3;i++)
        {
            if(fs[i]==0)
                index=i;
        }
        fr[index]=p[j];
        pf++;
    }
display();
```

```
    }
    printf("\n No of page faults:%d",pf+frsize);
    return 0;
}
void display()
{
    int i;
    printf("\n");
    for(i=0;i<3;i++)
    {
    printf("\t%d",fr[i]);
    }
}
```

c)OPTIMAL

```
#include<stdio.h>

int fr[3],n,m;

void display();

int main(){

	int i,j,page[20],fs[10];

	int max,found=0,lg[3],index,k,l;

	int flag1=0,flag2=0,pf=0;

	float pr;

	printf("Enter length of the reference string:");

	scanf("%d",&n);

	printf("Enter the reference string:");

	for(i=0;i<n;i++)

		scanf("%d",&page[i]);

	printf("Enter no.of frames:");

	scanf("%d",&m);

	for(i=0;i<m;i++)

		fr[i]=-1;pf=m;

	for(j=0;j<n;j++)

	{

		flag1=0;

		flag2=0;

		for(i=0;i<m;i++)

		{

			if(fr[i]==page[j])

			{

				flag1=1;

				flag2=1;
```

```
        break;
        }
    }
if(flag1==0){
    for(i=0;i<m;i++)
    {
        if(fr[i]==-1)
        {
            fr[i]=page[j];
            flag2=1;
            break;
        }
    }
}
if(flag2==0)
{
    for(i=0;i<m;i++)
        lg[i]==0;
    for(i=0;i<m;i++)
    {
        for(k=j+1;k<=n;k++)
        {
            if(fr[i]==page[k])
                {
                lg[i]=k-j;
                break;
                }
        }
    }
```

```
        found=0;
        for(i=0;i<m;i++)
        {
        if(lg[i]==0)
        {
            index=i;
            found=1;
            break;
        }
        }
        if(found==0)
        {
            max=lg[0];
            index=0;
            for(i=0;i<m;i++)
            {
                if(max<lg[i])
                {
                    max=lg[i];
                    index=i;
                    }
                }
            }
        fr[index]=page[j];
        pf++;
    }
    display();
}
```

```
printf("No.of page faults :%d\n",pf);
pr=(float)pf/n*100;
printf("page fault rate=%f\n",pr);
}

        void display()
        {
                int i;
                for(i=0;i<m;i++)
                        printf("%d\t",fr[i]);
                        printf("\n");
        }
```

Analyze the seek time for the following Disk scheduling algorithms –

1. FCFS; 2. SCAN; 3. LOOK

1)FCFS

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n,i,head,total=0;
    printf("enter the no.of requests:");
    scanf("%d",&n);
    int requests[n];
    printf("enter the request queue:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&requests[i]);
        }
    printf("enter the initial head positon:");
    scanf("%d",&head);
    printf("head movement order:\n");
    for(i=0;i<n;i++){
    printf("%d ->",head);
    total+=abs(requests[i]-head);
    head=requests[i];
    }
    printf("%d ->",head);
    printf("End\n");
    printf("total head movements %d\n",total);
    return 0;
    }
```

```
}
```

2)SCAN

```
#include<stdio.h>
#include<stdlib.h>
int main(){
        int n,i,head,direction,total=0,f;
        printf("Enter the no.o requests:");
        scanf("%d",&n);
        int requests[n];
        printf("Enter the request queue:\n");
        for(i=0;i<n;i++)
        {
                scanf("%d",&requests[i]);
                }
        printf("Enter the initial head position:");
        scanf("%d",&head);
        printf("Enter the direction(0 for left,1 for right):");
        scanf("%d",&direction);
        for(i=0;i<n-1;i++)
        {
                for(int j=0;j<n-i-1;j++)
                {
                        if(requests[j]>requests[j+1])
                        {
                                int temp=requests[j];
                                requests[j]=requests[j+1];
                                requests[j+1]=temp;
                        }
                }
        }
```

```
int current=head;
if(direction==1)
{
        for(i=0;i<n;i++)
        {
                if(requests[i]>=head)
                {
                        break;
                }
        }
}
else{
        for(i=n-1;i>=0;i--)
        {
                if(requests[i]<=head)
                {
                        break;
                }
        }
}
f=i;
printf("Head Movement Order:\n");
if(direction==1)
{
        for(;i<n;i++)
        {
                printf("%d->",current);
                total+=abs(requests[i]-current);
                current=requests[i];
```

```
    }
    if(current==requests[n-1])
    {
        printf("%d->",requests[n-1]);
        total+=abs(199-requests[n-1]);
        printf("199->");
        current=199;
        }
    for(i=f-1;i>=0;i--)
    {
        total+=abs(requests[i]-current);
        current=requests[i];
        printf("%d->",current);
        }
}
else{
for(;i>=0;i--)
{
    printf("%d->",current);
    total+=abs(requests[i]-current);
    current=requests[i];
}
printf("%d->",current);
if(current==requests[0])
{
    total+=abs(current-0);
    printf("0->");
    current=0;
}
```

```
		for(i=f+1;i<n;i++)
		{
			total+=abs(requests[i]-current);
			current=requests[i];
			printf("%d->",current);
		}
	}
	printf("End\n");
	printf("Total head movements:%d\n",total);
	return 0;
	}
```

c)LOOK

```
#include<stdio.h>
#include<stdlib.h>
int main(){
	int n,i,head,direction,total=0,f;
	printf("Enter the no.o requests:");
	scanf("%d",&n);
	int requests[n];
	printf("Enter the request queue:\n");
	for(i=0;i<n;i++)
	{
		scanf("%d",&requests[i]);
	}
	printf("Enter the initial head position:");
	scanf("%d",&head);
	printf("Enter the direction(0 for left,1 for right):");
	scanf("%d",&direction);
	for(i=0;i<n-1;i++)
	{
		for(int j=0;j<n-i-1;j++)
		{
			if(requests[j]>requests[j+1])
			{
				int temp=requests[j];
				requests[j]=requests[j+1];
				requests[j+1]=temp;
			}
```

```
            }
}
int current=head;
if(direction==1)
{
            for(i=0;i<n;i++)
            {
                        if(requests[i]>=head)
                        {
                                    break;
                        }
            }
}
else{
            for(i=n-1;i>=0;i--)
            {
                        if(requests[i]<=head)
                        {
                                    break;
                        }
            }
}
f=i;
printf("Head Movement Order:\n");
printf("%d->",current);
 if(direction==1)
{
            for(;i<n;i++)
            {
```

```
            total+=abs(requests[i]-current);
            current=requests[i];
            printf("%d->",current);



        }
        for(i=f-1;i>=0;i--)
        {
            total+=abs(requests[i]-current);
            current=requests[i];
            printf("%d->",current);
        }
    }
    else{
    for(;i>=0;i--)
    {

        total+=abs(requests[i]-current);
        current=requests[i];
        printf("%d->",current);
    }
    for(i=f+1;i<n;i++)
    {
        total+=abs(requests[i]-current);
        current=requests[i];
        printf("%d->",current);
    }
}
printf("End\n");
printf("Total head movements:%d\n",total);
return 0;
}
```