# Assign 1b

**Aim:-** write a program to implement an addressbook with options given below a) create addressbook b) view addressbook c) insert a record d) delete a record e) modify a record f) exit

# Code

Code

# Addressbook Assignment createBook()

{

echo

if [ -e addressbook.txt ] then

echo 'Address Book is already Created...' echo -e '\n'

else

touch addressbook.txt

echo 'Address Book Created...' echo -e '\n'

fi

}

insertBook()

{

while true do

echo -e '\n'

echo 'Enter your Full Name: ' read record

if [ '$record' == 'q' ] then

break fi

echo $record >> addressbook.txt echo -e '\n'

echo 'Record Inserted Successfully' echo -e '\n'

break done

}

displayBook()

{

if [ ! -e addressbook.txt ] then

echo -e '\n'

echo 'Address Book Does not Exit' else

cat addressbook.txt fi

}

searchBook()

{

echo -e '\n'

echo 'Enter the full name to search:' read data

if [ -f addressbook.txt ] then

result=$(grep -i "$data" addressbook.txt) if [ "$result" ]

then

echo -e '\n'

echo 'Record found successfully!' else

echo -e '\n'

echo 'Record not found' fi

else

echo -e '\n'

echo "Record not found" fi

}

deleteBook()

{

echo -e '\n'

echo "Enter the full name to delete:" read data

if [ -f addressbook.txt ] then

sed -i "/$data/d" addressbook.txt

sed -i "/^[[:space:]]\*$/d" addressbook.txt echo "Record deleted succesfully"

else

echo -e '\n'

echo "File not found" fi

}

modifyBook()

{

echo -e '\n'

echo "Enter Full Name which you want to modify: " read actual

result=$(grep -i "$actual" addressbook.txt) if [ "$result" ]

then

echo -e '\n'

echo "Enter new data: " read modified

sed -i "s/$actual/$modified/" addressbook.txt echo -e '\n'

echo "Record modified successfully..." else

echo -e '\n'

echo "Records not found" fi

}

while true do

echo -e '\n'

echo "Welcome To Menu" echo "1.Create Address Book" echo "2.Insert Data In Book"

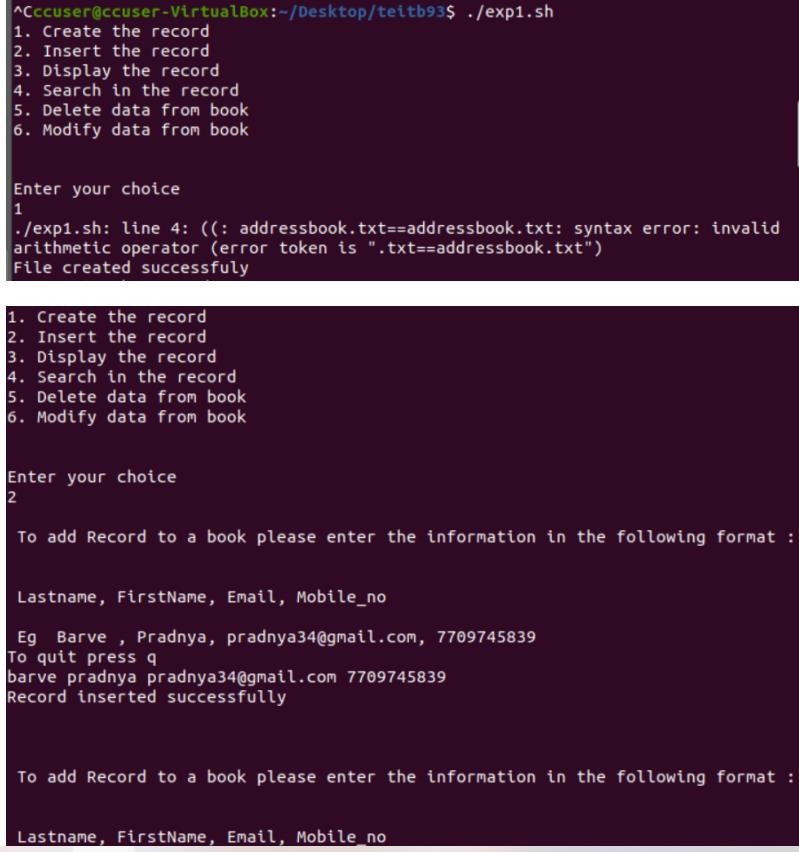
echo "3.Display Data From Book" echo "4.Search Data From Book" echo "5.Delete Data From Book" echo "6.Modify data From Book" echo "7.Exit"

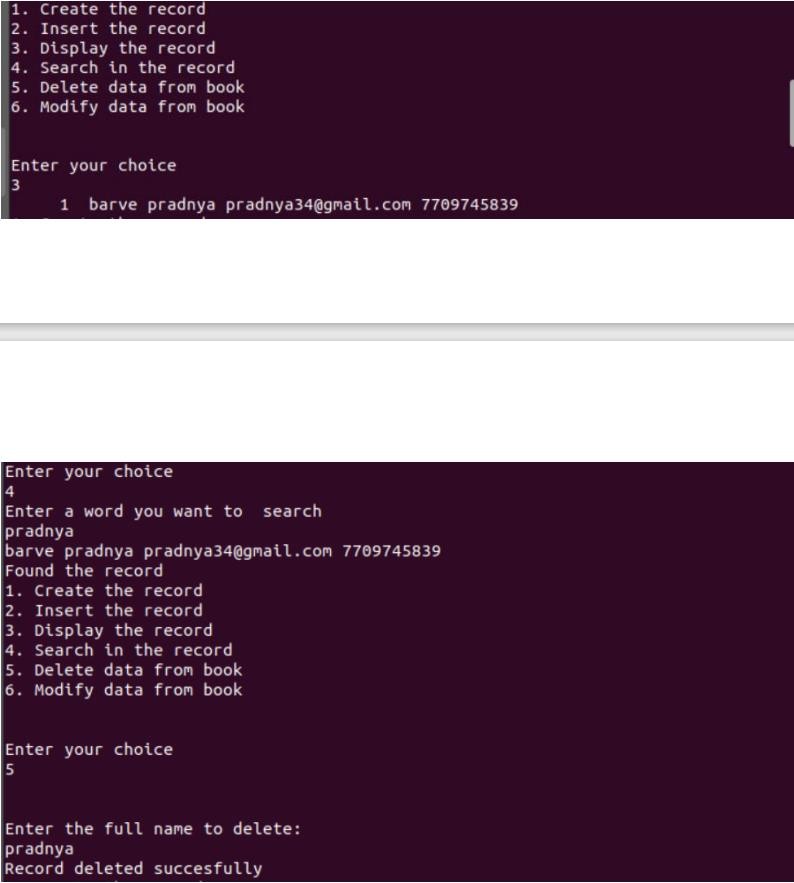
echo -e '\n'

echo "Enter Your Choice: " read ch

case $ch in 1)createBook;; 2)insertBook;; 3)displayBook;; 4)searchBook;; 5)deleteBook;; 6)modifyBook;; 7)exit;;

esac done **Output**





# Assign 2a

**Aim:-** Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

# Code Orphan

#include<stdio.h> #include<unistd.h>

#include<sys/types.h> // For fork() systemcall and pid\_t data type #define MAX 20

void quicksort(int a[],int,int); //prototype of Quick sort

void merge(int a[], int low, int mid, int high); //prototype of Merge sort void divide(int a[], int low, int high);

int main()

{

pid\_t pid; // Decleration of pid which will store process ID int a[MAX],n;

int i;

// Accepting Elements of an array printf("\n\tEnter the no. of elements: "); scanf("%d",&n);

printf("\n\tEnter the elements: \n"); for(i=0;i<n;i++)

{

printf("\t"); scanf("%d",&a[i]);

}

/\* =====Performing fork() system call==== \*/ pid=fork();

if(pid<0) // If Process not created successfully

{

printf("Error While creating a new process. !!!!!!");

}

else if(pid==0) // For Child process

{

wait(); // For Orphan process printf("\n\t==============Child process started=============");

printf("\n\tI am a child process with pid=%d and ppid=%d",getpid(),getppid());

quicksort(a,0,n-1); //Performing quick sort in child process printf("\n\n\tSorted array by quick sort:\n\t"); for(i=0;i<n;i++)

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

printf("\n\t==============Child process terminated=============\n");

}

else // For Parent process

{

printf("\n\t==============Parent process started=============");

printf("\n\n\tI am a parent process with pid=%d ",getpid()); divide(a, 0, n-1); //Performing merge sort in parent process printf("\n\n\tSorted array by merge sort:\n\t"); for(i=0;i<n;i++)

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

printf("\n\t==============Parent process terminated=============\n");

}

execl("/bin/ps","ps",NULL); return 0;

}

/\* ==== Definition of Quick Sort ====\*/ void quicksort(int a[MAX],int first,int last)

{

int pivot,j,i,temp; if(first<last)

{

i=first; j=last; pivot=first; while(i<j)

{

while(a[i]<=a[pivot] && i<last) i++;

while(a[j]>a[pivot]) j--;

if(i<j)

{

temp=a[i]; a[i]=a[j]; a[j]=temp;

}

}

temp=a[j]; a[j]=a[pivot]; a[pivot]=temp; quicksort(a,first,j-1);

quicksort(a,j+1,last);

}

}

/\* ==== Definition of Merge Sort ====\*/ void divide(int a[MAX], int low, int high)

{

if(low<high) // The array has atleast 2 elements

{

int mid = (low+high)/2;

divide(a, low, mid); // Recursion chain to sort first half of the array

divide(a, mid+1, high); // Recursion chain to sort second half of the array

merge(a, low, mid, high);

}

}

void merge(int a[MAX], int low, int mid, int high)

{

int i, j, k, m = mid-low+1, n = high-mid; int first\_half[m], second\_half[n];

for(i=0; i<m; i++) // Extract first half (already sorted) first\_half[i] = a[low+i];

for(i=0; i<n; i++) // Extract second half (already sorted) second\_half[i] = a[mid+i+1];

i=j=0;

k = low;

while(i<m || j<n) // Merge the two halves

{

if(i >= m)

{

a[k++] = second\_half[j++]; continue;

}

if(j >= n)

{

a[k++] = first\_half[i++]; continue;

}

if(first\_half[i] < second\_half[j]) a[k++] = first\_half[i++];

else

a[k++] = second\_half[j++];

}

}

**Zombie** #include<stdio.h> #include<unistd.h>

#include<sys/types.h> // For fork() systemcall and pid\_t data type #define MAX 20

void quicksort(int a[],int,int); //prototype of Quick sort

void merge(int a[], int low, int mid, int high); //prototype of Merge sort void divide(int a[], int low, int high);

int main()

{

pid\_t pid; // Decleration of pid which will store process ID int a[MAX],n;

int i;

// Accepting Elements of an array printf("\n\tEnter the no. of elements: ");

scanf("%d",&n);

printf("\n\tEnter the elements: \n"); for(i=0;i<n;i++)

{

printf("\t"); scanf("%d",&a[i]);

}

/\* =====Performing fork() system call==== \*/ pid=fork();

if(pid<0) // If Process not created successfully

{

printf("Error While creating a new process. !!!!!!");

}

else if(pid==0) // For Child process

{

printf("\n\t==============Child process started=============");

printf("\n\tI am a child process with pid=%d and ppid=%d",getpid(),getppid());

quicksort(a,0,n-1); //Performing quick sort in child process printf("\n\n\tSorted array by quick sort:\n\t"); for(i=0;i<n;i++)

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

printf("\n\t==============Child process terminated=============\n");

}

else // For Parent process

{

wait(); // For Zombie process

printf("\n\t==============Parent process started=============");

printf("\n\n\tI am a parent process with pid=%d ",getpid()); divide(a, 0, n-1); //Performing merge sort in parent process printf("\n\n\tSorted array by merge sort:\n\t"); for(i=0;i<n;i++)

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

printf("\n\t==============Parent process terminated=============\n");

}

execl("/bin/ps","ps",NULL); return 0;

}

/\* ==== Definition of Quick Sort ====\*/ void quicksort(int a[MAX],int first,int last)

{

int pivot,j,i,temp; if(first<last)

{

i=first; j=last; pivot=first; while(i<j)

{

while(a[i]<=a[pivot] && i<last) i++;

while(a[j]>a[pivot]) j--;

if(i<j)

{

temp=a[i]; a[i]=a[j]; a[j]=temp;

}

}

temp=a[j]; a[j]=a[pivot]; a[pivot]=temp; quicksort(a,first,j-1); quicksort(a,j+1,last);

}

}

/\* ==== Definition of Merge Sort ====\*/ void divide(int a[MAX], int low, int high)

{

if(low<high) // The array has atleast 2 elements

{

int mid = (low+high)/2;

divide(a, low, mid); // Recursion chain to sort first half of the array

divide(a, mid+1, high); // Recursion chain to sort second half of the array

merge(a, low, mid, high);

}

}

void merge(int a[MAX], int low, int mid, int high)

{

int i, j, k, m = mid-low+1, n = high-mid; int first\_half[m], second\_half[n];

for(i=0; i<m; i++) // Extract first half (already sorted) first\_half[i] = a[low+i];

for(i=0; i<n; i++) // Extract second half (already sorted) second\_half[i] = a[mid+i+1];

i=j=0;

k = low;

while(i<m || j<n) // Merge the two halves

{

if(i >= m)

{

a[k++] = second\_half[j++]; continue;

}

if(j >= n)

{

a[k++] = first\_half[i++]; continue;

}

if(first\_half[i] < second\_half[j]) a[k++] = first\_half[i++];

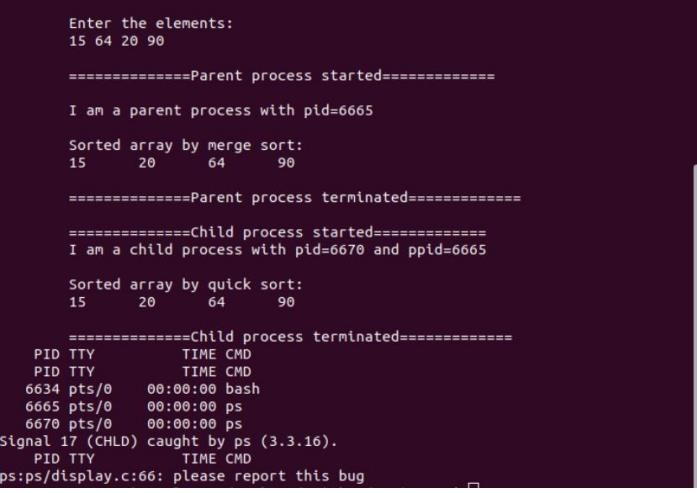
else

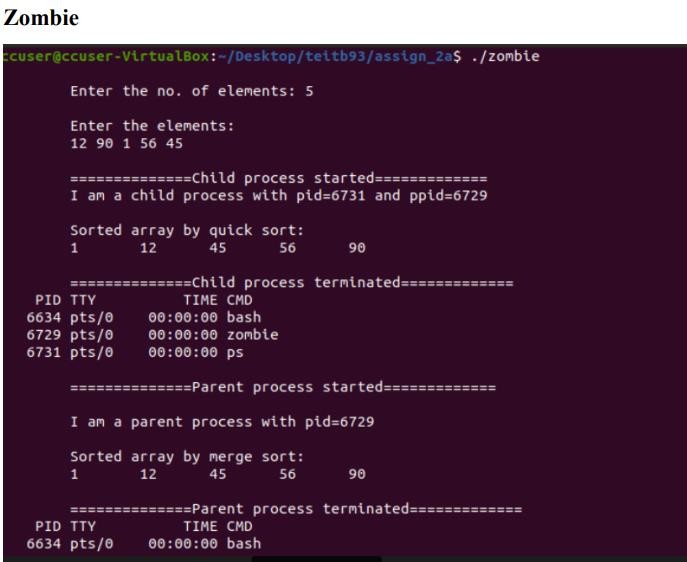
a[k++] = second\_half[j++];

}

}

# Output Orphan





**Assign 2b**

**Aim:-** Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order.

# Code Child

#include <bits/stdc++.h> #include <string.h> #include <sys/types.h> #include<sys/wait.h> #include <unistd.h>

#define MAX 20

int main(int argc, char \*argv[]){ int a[MAX];

for (int i = 0; i < argc; i++){ a[i] = atoi(argv[i]);

}

int rev[MAX];

for (int i = 0, j = argc-1; i > argc, j >= 0; i++, j--){ rev[i] = a[j];

}

cout << "\n The reversed array is: "; for (int i = 0; i < argc; i++){

printf("%d",rev[i]," ");

}

printf("\n");

}

# Execv

#include <bits/stdc++.h> #include <string.h> #include <sys/types.h> #include<sys/wait.h> #include <unistd.h>

#define MAX 20 using namespace std;

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; int i = (low - 1);

for (int j = low; j <= high - 1; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]); return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high);

}

}

int main(){

int a[MAX], n, i; char str[5];

char \*str1[MAX]; char ch;

pid\_t pid;

printf("Enter the number of elements: "); scanf("%d",&n);

printf("Enter the elements: "); for (int i = 0; i < n; i++){

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

}

pid = fork();

if (pid < 0){

printf("Error\n");

}

else if (pid>0){

quickSort(a, 0, n-1);

for (int i = 0; i < MAX; i++){ str1[i] = NULL;

}

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

sprintf(str, "%d", a[i]); str1[i] = strdup(str);

}

else{

}

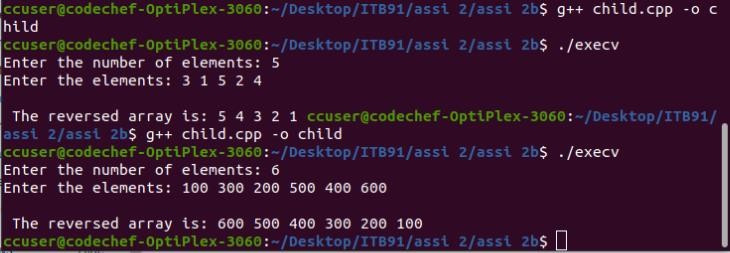
}

execve("child", str1, NULL);

return 0;

}

# Output



**Assign 3**

**Aim:-** Implement the C program for CPU Scheduling Algorithms: Shortest Job First(Preemptive) and Round Robin with different arrival time.

# Code Srjf code

#include <stdio.h>

int main()

{

int arrival\_time[10], burst\_time[10], temp[10]; int i, smallest, count = 0, time, limit;

double wait\_time = 0, turnaround\_time = 0, end;

float average\_waiting\_time, average\_turnaround\_time;

printf("\nEnter the Total Number of Processes: "); scanf("%d", &limit);

printf("\nEnter Details of %d Processes: ", limit);

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

{

printf("\nEnter Arrival Time: "); scanf("%d", &arrival\_time[i]); printf("Enter Burst Time:t"); scanf("%d", &burst\_time[i]); temp[i] = burst\_time[i];

}

burst\_time[9] = 9999;

for(time = 0; count != limit; time++)

{

smallest = 9;

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

{

if(arrival\_time[i] <= time && burst\_time[i] < burst\_time[smallest] && burst\_time[i] > 0)

{

smallest = i;

}

}

burst\_time[smallest]--; if(burst\_time[smallest] == 0)

{

count++;

end = time + 1;

wait\_time = wait\_time + end - arrival\_time[smallest] - temp[smallest]; turnaround\_time = turnaround\_time + end - arrival\_time[smallest];

}

}

average\_waiting\_time = wait\_time / limit; average\_turnaround\_time = turnaround\_time / limit; printf("Average Waiting Time: %lf", average\_waiting\_time);

printf("\nAverage Turnaround Time: %lf", average\_turnaround\_time); return 0;

}

**Round Robin code** #include<stdio.h> int main()

{

int i, limit, total = 0, x, counter = 0, time\_quantum;

int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10];

float average\_wait\_time, average\_turnaround\_time; printf("\nEnter Total Number of Processes: "); scanf("%d", &limit);

x = limit;

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

{

printf("\nEnter Details of Process[%d]\n", i + 1); printf("Arrival Time: ");

scanf("%d", &arrival\_time[i]); printf("Burst Time: "); scanf("%d", &burst\_time[i]); temp[i] = burst\_time[i];

}

printf("\nEnter Time Quantum: "); scanf("%d", &time\_quantum);

printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Timen"); for(total = 0, i = 0; x != 0;)

{

if(temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i]; temp[i] = 0;

counter = 1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum; total = total + time\_quantum;

}

if(temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);

wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i]; turnaround\_time = turnaround\_time + total - arrival\_time[i]; counter = 0;

}

if(i == limit - 1)

{

i = 0;

}

else if(arrival\_time[i + 1] <= total)

{ i++;

}

else

{

i = 0;

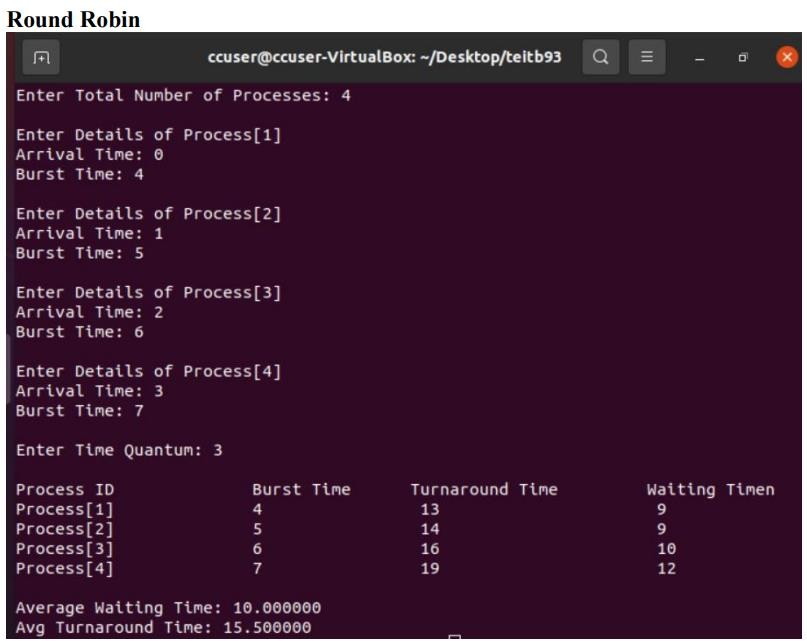
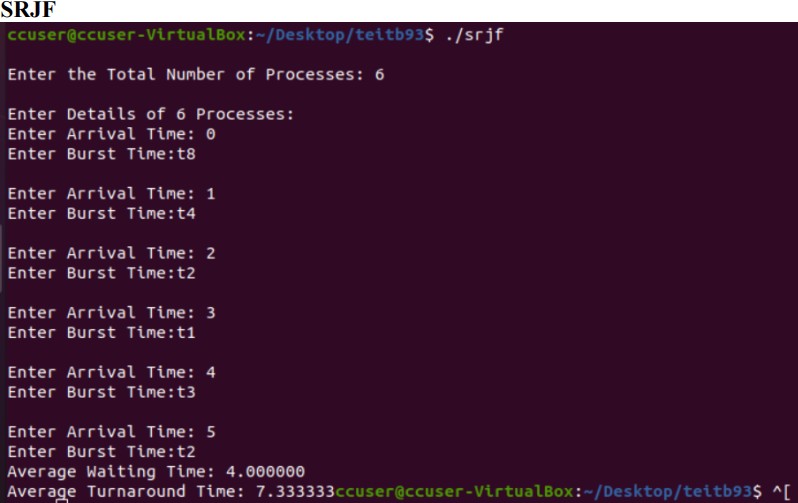
}

}

average\_wait\_time = wait\_time \* 1.0 / limit; average\_turnaround\_time = turnaround\_time \* 1.0 / limit; printf("\n\nAverage Waiting Time: %f", average\_wait\_time); printf("\nAvg Turnaround Time: %f\n", average\_turnaround\_time); return 0;

}

# Output



**Assign 5**

**Aim** :- Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

**Code** #include<stdio.h> void main() {

int k=0, output[10], d=0, t=0, ins[5], i, avail[5], allocated[10][5], need[10][5], MAX[10][5], pno, P[10], j, rz, count=0;

printf("\n Enter the number of resources : "); scanf("%d", &rz);

printf("\n enter the max instances of each resources\n"); for (i=0;i<rz;i++)

{

avail[i]=0; printf("%c= ",(i+97));

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

}

printf("\n Enter the number of processes : "); scanf("%d", &pno);

printf("\n Enter the allocation matrix \n "); for (i=0;i<rz;i++)

printf(" %c",(i+97)); printf("\n");

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

{

P[i]=i;

printf("P[%d] ",P[i]); for (j=0;j<rz;j++)

{

scanf("%d",&allocated[i][j]); avail[j]+=allocated[i][j];

}

}

printf("\nEnter the MAX matrix \n "); for (i=0;i<rz;i++)

{

printf(" %c",(i+97));

avail[i]=ins[i]-avail[i];

}

printf("\n");

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

{

printf("P[%d] ",i); for (j=0;j<rz;j++)

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

}

printf("\n");

A: d=-1;

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

{

count=0; t=P[i];

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

{

need[t][j] = MAX[t][j]-allocated[t][j]; if(need[t][j]<=avail[j])

count++;

}

if(count==rz)

{

output[k++]=P[i]; for (j=0;j<rz;j++)

avail[j]+=allocated[t][j];

} else

P[++d]=P[i];

}

if(d!=-1)

{

pno=d+1; goto A;

}

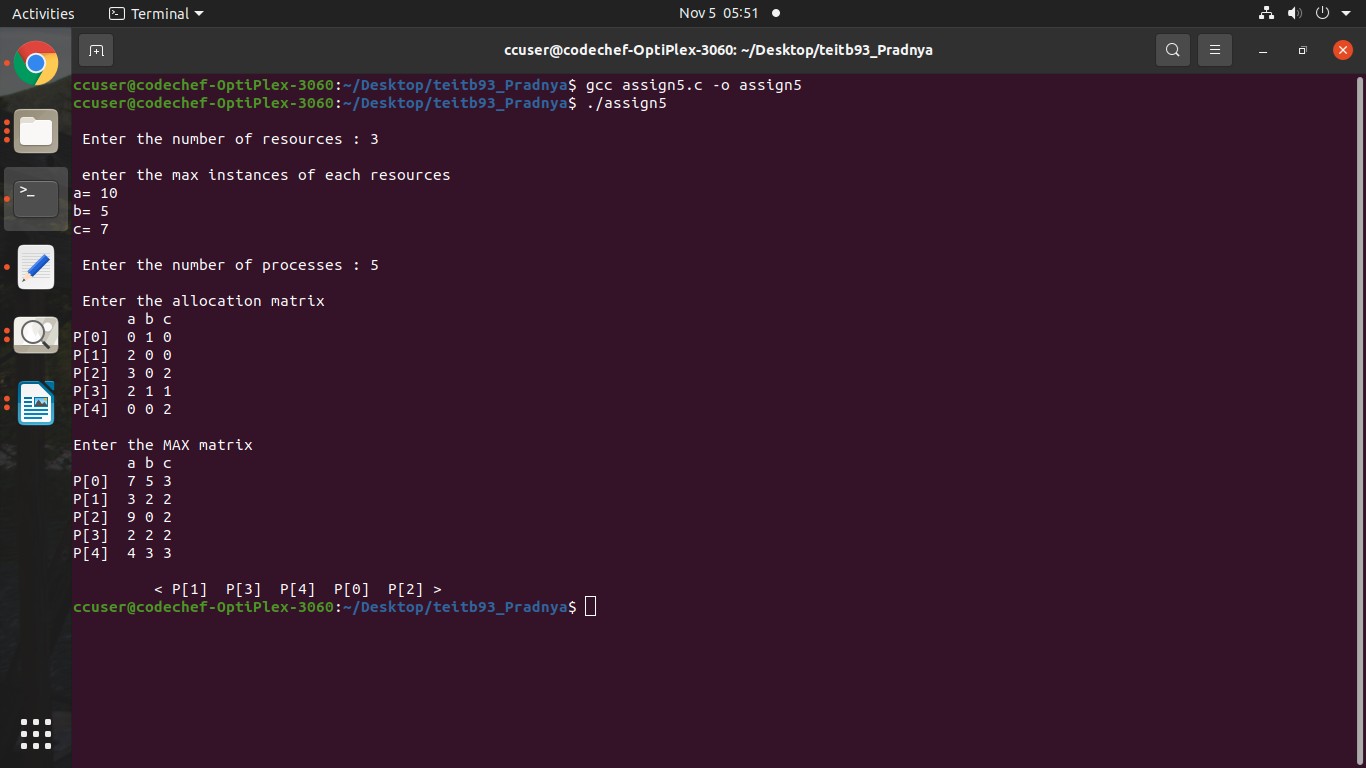
printf("\t <"); for (i=0;i<k;i++)

printf(" P[%d] ",output[i]); printf(">");

printf("\n");

}

# Output



**Assign 6**

**Aim :-** Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.

**Code** #include<stdio.h> int n,nf;

int in[100]; int p[50]; int hit=0; int i,j,k;

int pgfaultcnt=0; void getData()

{

printf("\nEnter length of page reference sequence:"); scanf("%d",&n); printf("\nEnter the page reference sequence:"); for(i=0; i<n; i++) scanf("%d",&in[i]);

printf("\nEnter no of frames:"); scanf("%d",&nf);

}

void initialize()

{

pgfaultcnt=0; for(i=0; i<nf; i++) p[i]=9999;

}

int isHit(int data)

{

hit=0;

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

{

if(p[j]==data)

{

hit=1; break;

}

}

return hit;

}

int getHitIndex(int data)

{

int hitind;

for(k=0; k<nf; k++)

{

if(p[k]==data)

{

hitind=k; break;

}

}

return hitind;

}

void dispPages()

{

for (k=0; k<nf; k++)

{

if(p[k]!=9999)

printf(" %d",p[k]);

}

}

void dispPgFaultCnt()

{

printf("\nTotal no of page faults:%d",pgfaultcnt);

}

void fifo()

{

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

{

printf("\nFor %d :",in[i]); if(isHit(in[i])==0)

{

}

else

for(k=0; k<nf-1; k++) p[k]=p[k+1];

p[k]=in[i]; pgfaultcnt++; dispPages();

printf("No page fault");

}

dispPgFaultCnt();

}

void optimal()

{

initialize(); int near[50];

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

{

printf("\nFor %d :",in[i]); if(isHit(in[i])==0)

{

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

{

int pg=p[j]; int found=0;

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

{

if(pg==in[k])

{

}

else

near[j]=k; found=1; break;

found=0;

}

if(!found) near[j]=9999;

}

int max=-9999; int repindex; for(j=0; j<nf; j++)

{

if(near[j]>max)

{

max=near[j]; repindex=j;

}

}

}

else

p[repindex]=in[i]; pgfaultcnt++; dispPages();

printf("No page fault");

}

dispPgFaultCnt();

}

void lru()

{

initialize(); int least[50];

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

{

printf("\nFor %d :",in[i]); if(isHit(in[i])==0)

{

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

{

int pg=p[j]; int found=0; for(k=i-1; k>=0; k--)

{

if(pg==in[k])

{

}

else

least[j]=k; found=1; break;

found=0;

}

if(!found) least[j]=-9999;

}

int min=9999; int repindex;

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

{

if(least[j]<min)

{

min=least[j]; repindex=j;

}

}

}

else

p[repindex]=in[i]; pgfaultcnt++; dispPages();

printf("No page fault!");

}

dispPgFaultCnt();

}

void lfu()

{

int usedcnt[100];

int least,repin,sofarcnt=0,bn; initialize(); for(i=0; i<nf; i++)

usedcnt[i]=0; for(i=0; i<n; i++)

{

printf("\n For %d :",in[i]); if(isHit(in[i]))

{

}

else

int hitind=getHitIndex(in[i]); usedcnt[hitind]++; printf("No page fault!");

{

pgfaultcnt++; if(bn<nf)

{

}

else

{

p[bn]=in[i]; usedcnt[bn]=usedcnt[bn]+1; bn++;

least=9999;

for(k=0; k<nf; k++) if(usedcnt[k]<least)

{

least=usedcnt[k]; repin=k;

}

p[repin]=in[i]; sofarcnt=0;

for(k=0; k<=i; k++) if(in[i]==in[k]) sofarcnt=sofarcnt+1; usedcnt[repin]=sofarcnt;

} dispPages();

}

}

dispPgFaultCnt();

}

void secondchance()

{

int usedbit[50]; int victimptr=0; initialize(); for(i=0; i<nf; i++) usedbit[i]=0; for(i=0; i<n; i++)

{

printf("\nFor %d:",in[i]); if(isHit(in[i]))

{

}

else

{

printf("No page fault!"); int hitindex=getHitIndex(in[i]); if(usedbit[hitindex]==0) usedbit[hitindex]=1;

pgfaultcnt++; if(usedbit[victimptr]==1)

{

do

{

usedbit[victimptr]=0; victimptr++; if(victimptr==nf)

victimptr=0;

}

while(usedbit[victimptr]!=0);

}

if(usedbit[victimptr]==0)

{

p[victimptr]=in[i]; usedbit[victimptr]=1; victimptr++;

}

dispPages();

}

if(victimptr==nf) victimptr=0;

}

dispPgFaultCnt();

}

int main()

{

int choice; while(1)

{

printf("\nPage Replacement Algorithms\n1.Enter data\n2.FIFO\n3.Optimal\n4.LRU\n5.LFU\n6.Second

Chance\n7.Exit\nEnter your choice:"); scanf("%d",&choice);

switch(choice)

{

case 1: getData(); break;

case 2: fifo(); break;

case 3:optimal(); break;

case 4: lru(); break;

case 5: lfu(); break;

case 6: secondchance(); break;

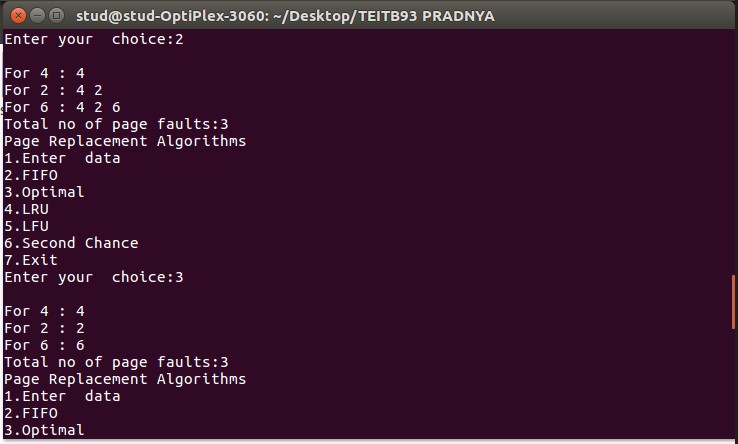
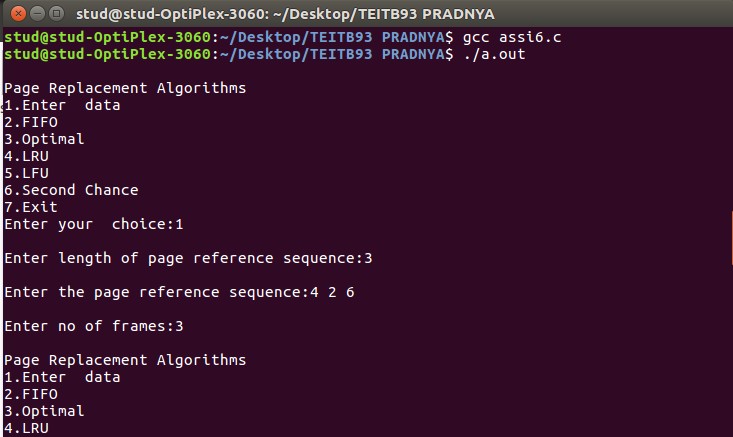
default: return 0; break;

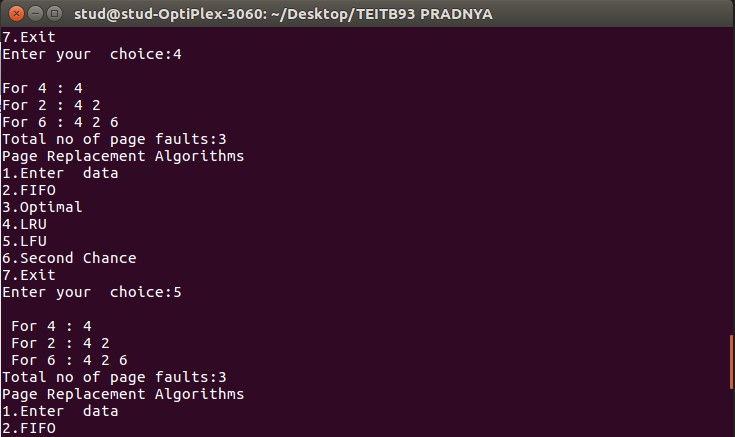
}

}

}

# Output







**Assign 7a**

**Aim**:- FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

# Code Sender

#include <stdio.h> #include <fcntl.h> #include <stdlib.h> #include <string.h>

#include <fcntl.h> // for open #include <unistd.h> // for close #include<sys/types.h> #include<string.h>

#include <sys/stat.h>

//sender

int main()

{

int res,n;

char buffer[100];

res = mkfifo(“fifo1”, 0666); char str1[100]; printf(“fifo1 created\n”);

printf(“Enter string to write :”); fgets(str1,100,stdin);

res = open(“fifo1”,O\_RDWR); write(res,str1,100);

res = open(“fifo2”, O\_RDWR); read(res,buffer,100);

printf(“printing data received from fifo2 %s\n”,buffer);

return 0;

}

# Receiver

#include <stdio.h> #include <fcntl.h> #include <stdlib.h> #include <string.h>

#include <fcntl.h> // for open #include <unistd.h> // for close #include<sys/types.h> #include <sys/stat.h> #include<string.h>

//sender

int count\_char(char \*str)

{

int I = 0, space = 0; while (str[i] != ‘\0’)

{

if (str[i] != ‘ ‘ && str[i] != ‘\n’)

{

space++;

}

i++;

}

return space;

}

int count\_word(char \*str)

{

int I,wrd; i=0;

wrd=0;

while(str[i] != ‘\0’){

if(str[i] == ‘ ‘ || str[i] == ‘\n’ || str[i] == ‘t’){ wrd++;

} i++;

}

return (wrd);

}

int count\_lines(char \*str)

{

int I = 0, lines = 0; while (str[i] != ‘\0’)

{

if (str[i] == ‘\n’)

{

lines++;

}

i++;

}

return lines;

}

char\* int\_to\_string(int num){ char \*str = (char\*)malloc(10); 44print(str,”%d”,num); return str;

}

int main()

{

int res,n,ct1,ct2,res2,ct3; char buffer[100];

char str[100]; FILE \*file;

char \*tt1,\*tt2,\*tt3;

char cha[50] = “character are: “; char words[50] = “ and words are “; char lines[50] = “ lines are :”;

res = mkfifo(“fifo2”, 0666); printf(“fifo2 created\n”);

res = open(“fifo1”, O\_RDWR); read(res,buffer,100);

ct1 = count\_char(buffer); ct2 = count\_word(buffer); ct3 = count\_lines(buffer);

res2 = open(“fifo2”,O\_RDWR);

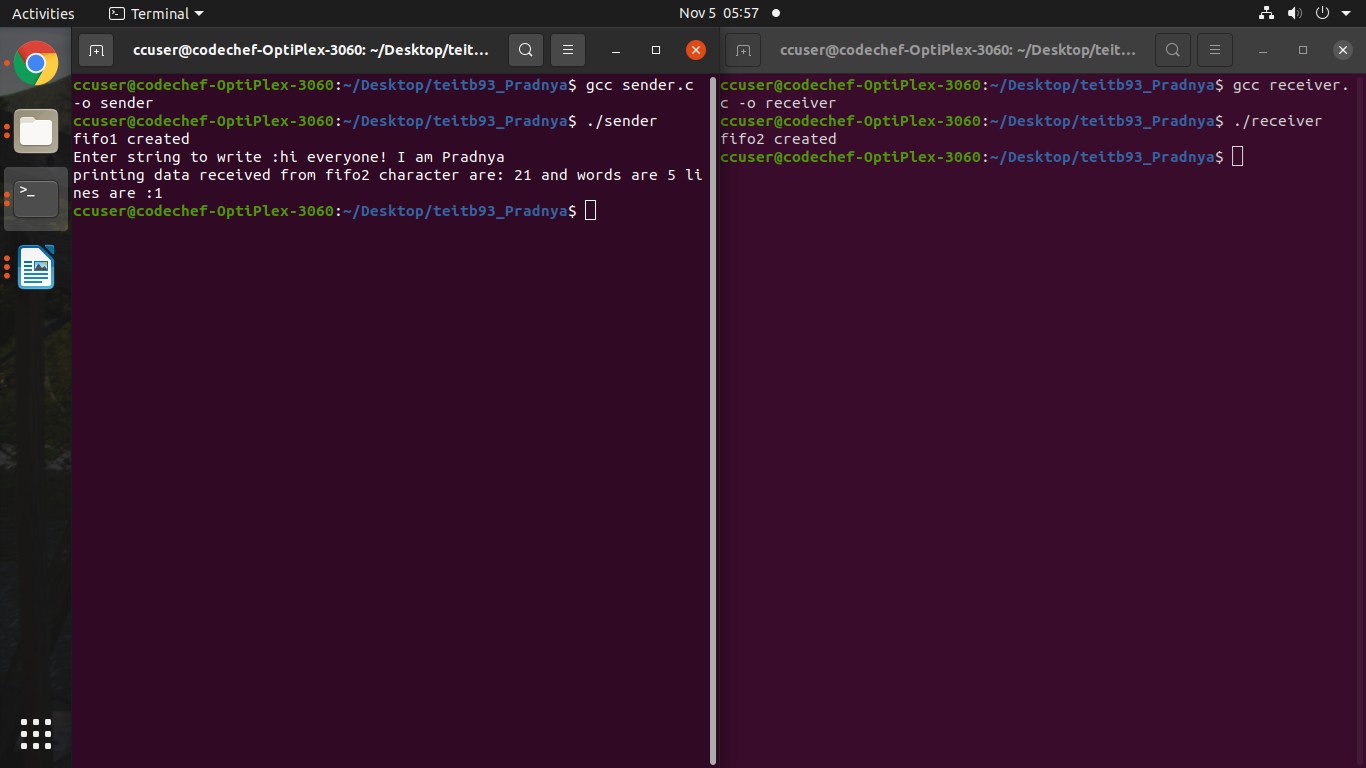
// write(res2,”KILLO”,20);

//strcat(cha,words); tt1 = int\_to\_string(ct1); tt2 = int\_to\_string(ct2); tt3 = int\_to\_string(ct3); strcat(cha,tt1); strcat(cha,words); strcat(cha,tt2); strcat(cha,lines); strcat(cha,tt3); write(res2,cha,100);

// write(res2,tt2,100); return 0;

}

# Output



**Assign 7b**

**Aim:-** Interprocess Communication using Shared Memory using System V.

Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

# Code Client

#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 the 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);

}

# Server

#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;

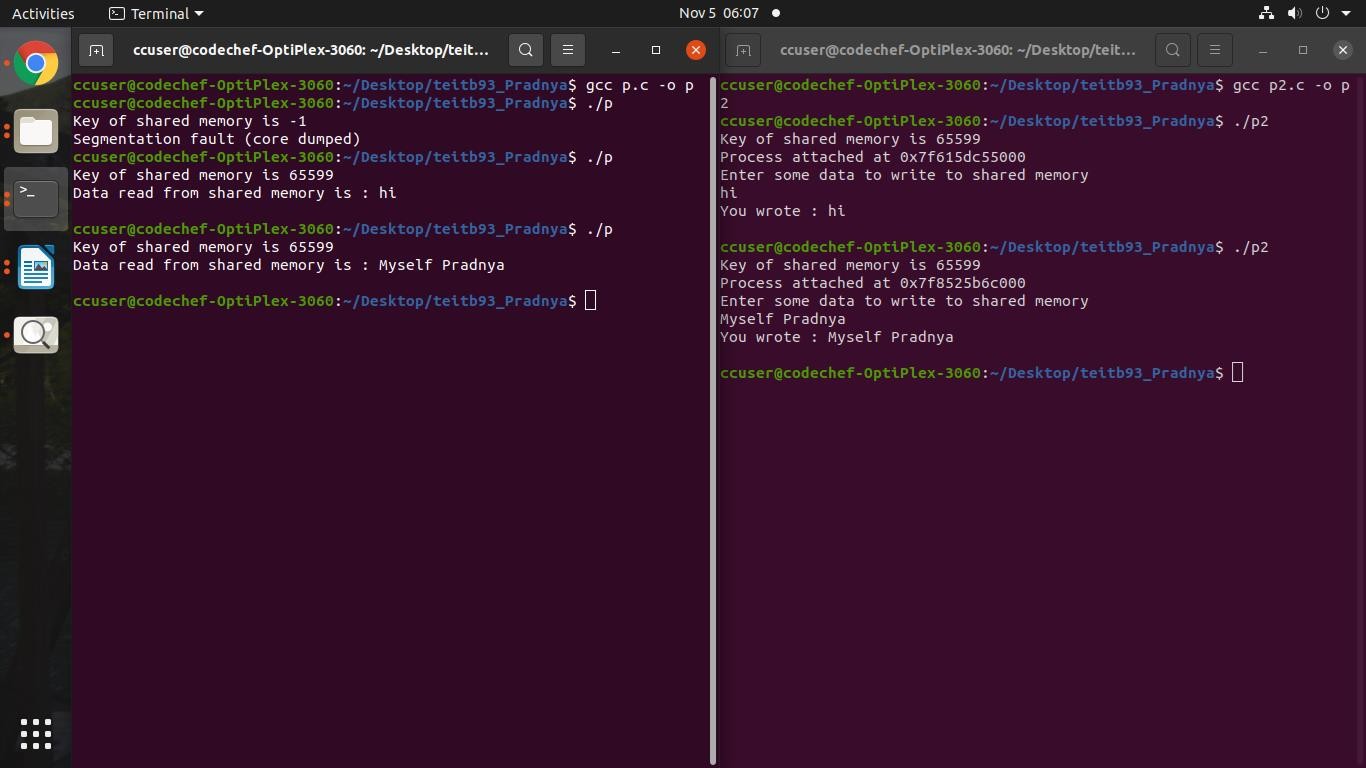
shmid=shmget((key\_t)2345, 1024, 0666); printf("Key of shared memory is %d\n",shmid);

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);

}

# Output



**Assign 8**

**Aim:-** Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.

# SSTF

**Code** #include<stdio.h> #include<stdlib.h> int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial,count=0; printf("Enter the number of Requests\n"); scanf("%d",&n);

printf("Enter the Requests sequence\n"); for(i=0;i<n;i++)

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

printf("Enter initial head position\n"); scanf("%d",&initial);

while(count!=n)

{

int min=1000,d,index; for(i=0;i<n;i++)

{

d=abs(RQ[i]-initial); if(min>d)

{

min=d; index=i;

}

}

TotalHeadMoment=TotalHeadMoment+min;

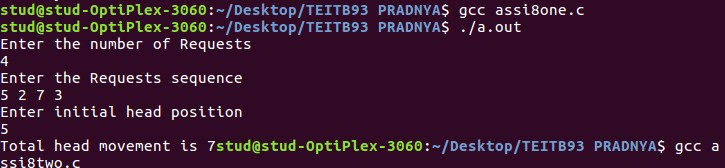
initial=RQ[index]; RQ[index]=1000; count++;

}

printf("Total head movement is %d",TotalHeadMoment); return 0;

}

# Output



**SCAN**

# Code

#include <stdio.h> #include <stdlib.h> int main(){

int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move; printf("Enter the number of Requests\n");

scanf("%d", &n);

printf("Enter the Requests sequence\n"); for (i = 0; i < n; i++)

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

printf("Enter initial head position\n"); scanf("%d", &initial);

printf("Enter total disk size\n"); scanf("%d", &size);

printf("Enter the head movement direction for high 1 and for low 0\n"); scanf("%d", &move);

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

for (j = 0; j < n - i - 1; j++){

if (RQ[j] > RQ[j + 1]){

int temp; temp = RQ[j];

RQ[j] = RQ[j + 1];

RQ[j + 1] = temp;

}

}

}

int index;

for (i = 0; i < n; i++){ if (initial < RQ[i]){

index = i; break;

}

}

if (move == 1){

for (i = index; i < n; i++){

TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial); initial = RQ[i];

}

TotalHeadMoment = TotalHeadMoment + abs(size - RQ[i - 1] - 1); TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0);

initial = 0;

for (i = 0; i < index; i++){

TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial); initial = RQ[i];

}

}

else{

for (i = index - 1; i >= 0; i--){

TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial); initial = RQ[i];

}

TotalHeadMoment = TotalHeadMoment + abs(RQ[i + 1] - 0); TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0); initial = size - 1;

for (i = n - 1; i >= index; i--){

TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial); initial = RQ[i];

}

}

printf("Total head movement is %d", TotalHeadMoment); return 0;

}

# Output



**Assign 9 Aim:-** To implement and simulate the MFT algorithm.

**Code** #include<stdio.h> #include<stdlib.h> int main()

{

int ms, bs, nob, ef,n, mp[10],tif=0; int i,p=0;

printf("Enter the total memory available (in Bytes) -- "); scanf("%d",&ms);

printf("Enter the block size (in Bytes) -- "); scanf("%d", &bs);

nob=ms/bs; ef=ms - 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\tMEMORY REQUIRED\t ALLOCATED\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%d",bs-mp[i]); tif = tif + bs-mp[i];

p++;

}

}

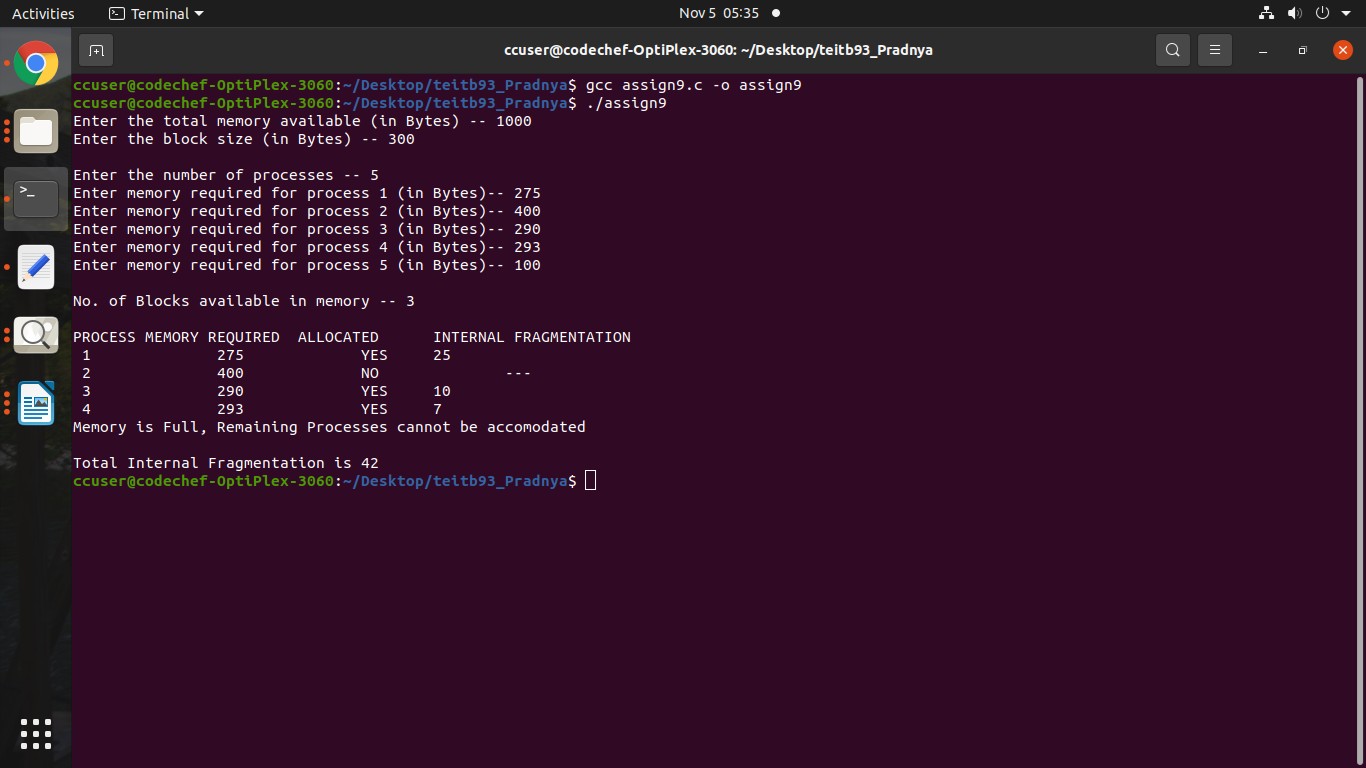
if(i<n)

printf("\nMemory is Full, Remaining Processes cannot be accomodated"); printf("\n\nTotal Internal Fragmentation is %d",tif);

printf("\nTotal External Fragmentation is %d",ef);

}

# Output



**Assign 10 Aim:-** To write a program to simulate the MVT algorithm **Code**

#include<stdio.h> #include<stdlib.h> int main()

{

int ms,mp[10],i, temp,n=0; char ch = 'y';

printf("\nEnter the total memory available (in Bytes)-- "); scanf("%d",&ms);

temp=ms; 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", ms);

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",ms-temp); printf("\nTotal External Fragmentation is %d",temp);

}

# Output

