CSA0462-OPERATING SYSTEMS FOR QUANTUM COMPUTER

## 

# **NAME:K.SANNIHITHA REDDY**

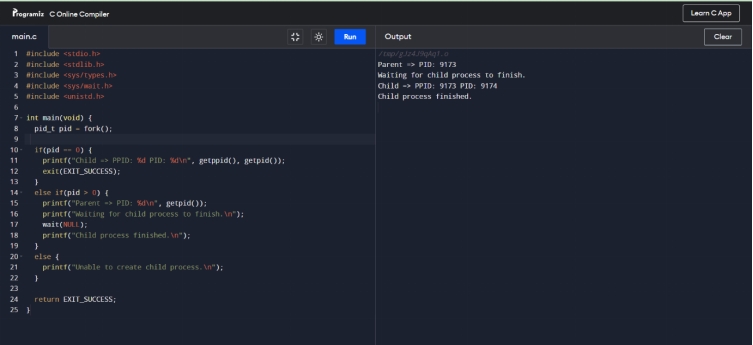
# **REG NO:192111413**

# **#PROGRAM 1**

**AIM**: Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

**Algorithm:**

1. fork() system call is used to create child processes in a C program. The fork() system function is defined in the headers sys/types.h and unistd.h..
2. To finish a child process, the exit() system call is used in the child process.
3. The wait() function is defined in the header sys/wait.h and the exit() function is defined in the header stdlib.h.
4. I used fork() to create a child process from the main/parent process. Then, I printed the PID (Process ID) and PPID (Parent Process ID) from child and parent process.
5. On the parent process wait(NULL) is used to wait for the child process to finish.
6. On the child process, exit() is used to finish the child process. As you can see, the PID of the parent process is the PPID of the child process.
7. So, the child process 9173 belongs to the parent process 9174
8. **Program:**
9. #include <stdio.h>
10. #include <stdlib.h>
11. #include <sys/types.h>
12. #include <sys/wait.h>
13. #include <unistd.h>
15. int main(void) {
16. pid\_t pid = fork();
18. if(pid == 0) {
19. printf("Child => PPID: %d PID: %d\n", getppid(), getpid());
20. exit(EXIT\_SUCCESS);
21. }
22. else if(pid > 0) {
23. printf("Parent => PID: %d\n", getpid());
24. printf("Waiting for child process to finish.\n");
25. wait(NULL);
26. printf("Child process finished.\n");
27. }
28. else {
29. printf("Unable to create child process.\n");
30. }
32. return EXIT\_SUCCESS;
33. }


37. **Output :**
38. Parent => PID: 18432
39. Waiting for child process to finish.
40. Child => PPID: 18432 PID: 18433
41. Child process finished
42. 

**#PROGRAM** 2

**Aim:**

Identify the system calls to copy the content of one file to another and illustrate the same using a C program

**Algorithm:**

The header files required are:

* Stdio.h for printf function
* unistd.h for open, close, read, and write system calls
* We have used pointers in the program.
* The pointer in C language is a variable which stores the address of another variable.
* This variable can be of type int, char, array, function, or any other pointer
* We have used pointers in the program.

**Program:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

FILE \*f1,\*f2;

char filename[100],c;

f1=fopen("D:\DEVC++\test2.c","r");

f2=fopen("D:\DEVC++\test1.txt","w");

c=fgetc(f1);

while(c!=EOF)

{

fputc(c,f2);

c=fgetc(f1);

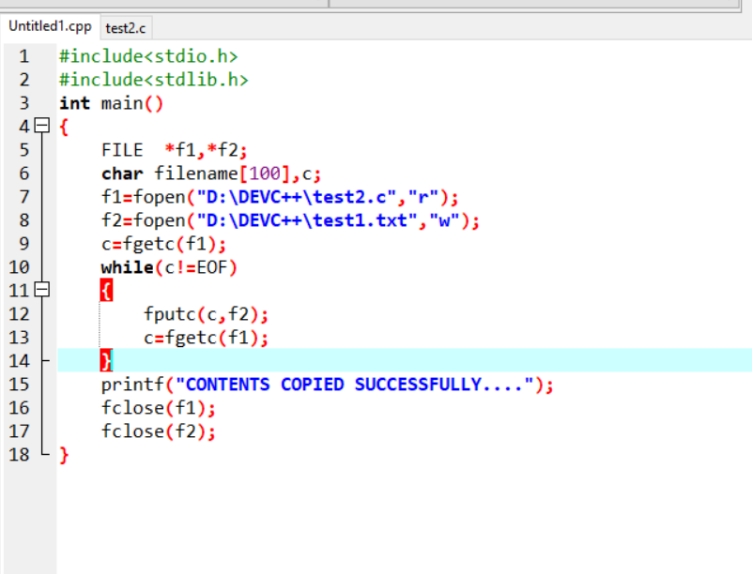
}

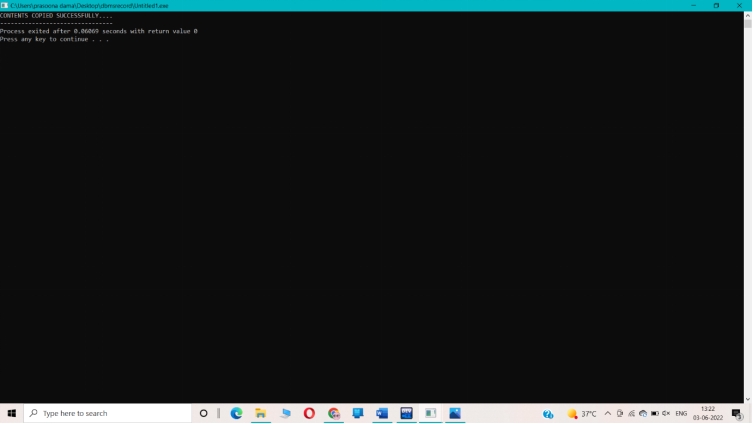
printf("CONTENTS COPIED SUCCESSFULLY....");

fclose(f1);

fclose(f2);

}

**output:** 



**Result:**

copied the content of one file to another file by using system call in c programing.

# **#PROGRAM 3**

##### AIM:

##### **To** **Write the C Program for CPU scheduling algorithm using FCFS**

##### ALGORITHM:

1: Inside the structure declare the variables.

2: Declare the variable i, j as integer, totwtime and totttime is equal to zero.

3: Get the value of „n‟ assign pid as I and get the value of p[i].btime.

4: Assign p[0] wtime as zero and tot time as btime and inside the loop calculate wait time and turnaround time.

5: Calculate total wait time and total turnaround time by dividing by total number of process. Step 6: Print total wait time and total turnaround time.

7: Stop the program.

**Program:**

#include<stdio.h>

int main()

{

int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;

printf("Enter total number of processes(maximum 20):");

scanf("%d",&n);

printf("\nEnter Process Burst Time\n");

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

{

printf("P[%d]:",i+1);

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

}

wt[0]=0;for(i=1;i<n;i++)

{

wt[i]=0;

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

wt[i]+=bt[j];

}

printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

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

{

tat[i]=bt[i]+wt[i];

avwt+=wt[i];

avtat+=tat[i];

printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);

}

avwt/=i;

avtat/=i;

printf("\n\nAverage Waiting Time:%d",avwt);

printf("\nAverage Turnaround Time:%d",avtat);

return 0;

}

**Output:**

Enter total number of processes(maximum 20):3

Enter Process Burst Time

P[1]:4

P[2]:5

P[3]:6

Process Burst Time Waiting Time Turnaround Time

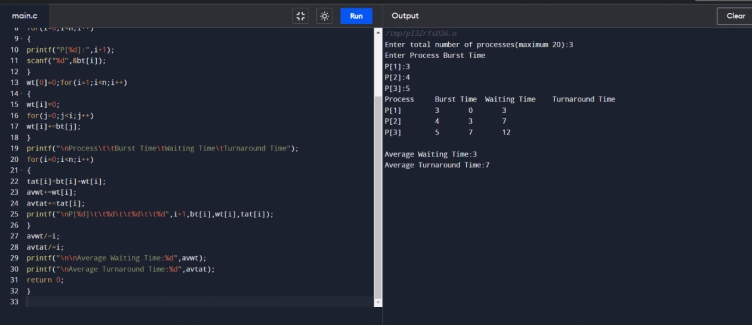
P[1] 4 0 4

P[2] 5 4 9

P[3] 6 9 15

Average Waiting Time:4

Average Turnaround Time:9



**RESULT:**

the FCFS algorithm for CPU scheduling has been executed successfully.

#PROGRAM 4

AIM:***Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next***

##### ALGORITHM:

Step 1: Inside the structure declare the variables.

Step 2: Declare the variable i,j as integer,totwtime and totttime is equal to zero.

Step 3: Get the value of „n‟ assign pid as I and get the value of p[i].btime.

Step 4: Assign p[0] wtime as zero and tot time as btime and inside the loop calculate wait time and turnaround time.

Step 5: Calculate total wait time and total turnaround time by dividing by total number of process. Step 6: Print total wait time and total turnaround time.

Step 7: Stop the program.

**PROGRAM:**

#include<stdio.h>

void main()

{

int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;

float avg\_wt,avg\_tat;

printf("Enter number of process:");

scanf("%d",&n);

printf("\nEnter Burst Time:\n");

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

{printf("p%d:",i+1);

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

p[i]=i+1;

}

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

{

pos=i;

for(j=i+1;j<n;j++)

{

if(bt[j]<bt[pos])

pos=j;

}

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=0;

for(j=0;j<i;j++)wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=(float)total/n;

total=0;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

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

{

tat[i]=bt[i]+wt[i];

total+=tat[i];

printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=(float)total/n;

printf("\n\nAverage Waiting Time=%f",avg\_wt);

printf("\nAverage Turnaround Time=%f\n",avg\_tat);

}

**OUTPUT:**

Enter number of process:4

Enter Burst Time:

p1:5

p2:6

p3:7

p4:8

Process Burst Time Waiting Time Turnaround Time

p1 5 0 5

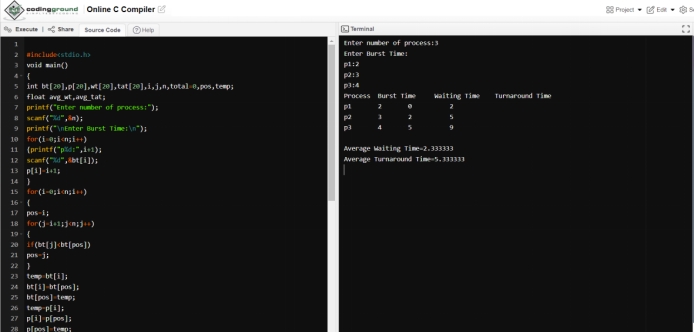
p2 6 5 11

p3 7 11 18

p4 8 18 26

Average Waiting Time=8.500000

Average Turnaround Time=15.000000



**RESULT:**

Hence the SJF algorithm for CPU scheduling has been executed successfully.

#PROGRAM 5

**AIM:**

Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.

**ALGORITM:**

1. Start.  
2. Get the values of resources and processes.  
3. Get the avail value.  
4. After allocation find the need value.  
5. Check whether its possible to allocate.  
6. If it is possible then the system is in safe state.  
7. Else system is not in safety state  
8. Stop .

PROGRAM:

#include<stdio.h>

void main()

{

int n,r,i,j,k,p,u=0,s=0,m;

int block[10],run[10],active[10],newreq[10];

int max[10][10],resalloc[10][10],resreq[10][10];

int totalloc[10],totext[10],simalloc[10];

//clrscr();

printf("Enter the no of processes:");

scanf("%d",&n);

printf("Enter the no ofresource classes:");

scanf("%d",&r);

printf("Enter the total existed resource in each class:");

for(k=1; k<=r; k++)

scanf("%d",&totext[k]);

printf("Enter the allocated resources:");

for(i=1; i<=n; i++)

for(k=1; k<=r; k++)

scanf("%d",&resalloc);

printf("Enter the process making the new request:");

scanf("%d",&p);

printf("Enter the requested resource:");

for(k=1; k<=r; k++)

scanf("%d",&newreq[k]);

printf("Enter the process which are n blocked or running:");

for(i=1; i<=n; i++)

{

if(i!=p)

{

printf("process %d:\n",i+1);

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

}

}

block[p]=0;

run[p]=0;

for(k=1; k<=r; k++)

{

j=0;

for(i=1; i<=n; i++)

{

totalloc[k]=j+resalloc[i][k];

j=totalloc[k];

}

}

for(i=1; i<=n; i++)

{

if(block[i]==1||run[i]==1)

active[i]=1;

else

active[i]=0;

}

for(k=1; k<=r; k++)

{

resalloc[p][k]+=newreq[k];

totalloc[k]+=newreq[k];

}

for(k=1; k<=r; k++)

{

if(totext[k]-totalloc[k]<0)

{

u=1;

break;

}

}

if(u==0)

{

for(k=1; k<=r; k++)

simalloc[k]=totalloc[k];

for(s=1; s<=n; s++)

for(i=1; i<=n; i++)

{

if(active[i]==1)

{

j=0;

for(k=1; k<=r; k++)

{

if((totext[k]-simalloc[k])<(max[i][k]-resalloc[i][k]))

{

j=1;

break;

}

}

}

if(j==0)

{

active[i]=0;

for(k=1; k<=r; k++)

simalloc[k]=resalloc[i][k];

}

}

m=0;

for(k=1; k<=r; k++)

resreq[p][k]=newreq[k];

printf("Deadlock willn't occur");

}

else

{

for(k=1; k<=r; k++)

{

resalloc[p][k]=newreq[k];

totalloc[k]=newreq[k];

}

printf("Deadlock will occur");

}

}

**OUTPUT:**

**Enter the no of processes:4**

**Enter the no ofresource classes:3**

**Enter the total existed resource in each class:3 2 2**

**Enter the allocated resources: 1 0 0 0 5 0 0**

**9**

**6**

**5**

**4**

**4**

**3**

**4**

**Enter the process making the new request:4**

**3**

**Enter the requested resource:2**

**4**

**Enter the process which are n blocked or running:process 2:**

**3**

**3**

**process 3:**

**2**

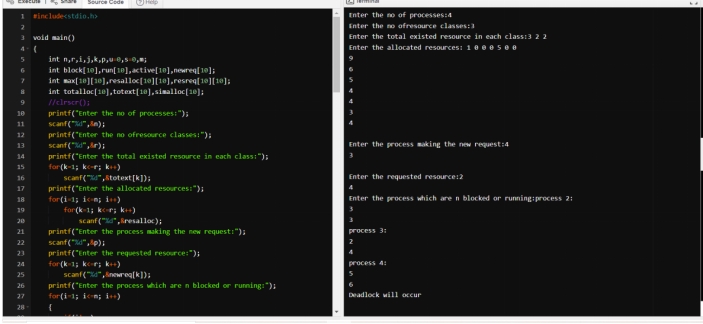
**4**

**process 4:**

**5**

**6**

**Deadlock will occur**



**RESULT:**

The deadlock avoidance concept by simulating Banker’s algorithm with C is successfully executed

**#PROGRAM 6**

**AIM:**

Construct a C program to simulate producer-consumer problem

using semaphores.

**ALGORITM:**

we need two counting semaphores – Full and Empty.“Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

**Semaphore :** A semaphore S is an integer variable that can be accessed only through two standard operations :

* wait() - The wait() operation reduces the value of semaphore by 1.
* signal() - The signal() operation increases its value by 1.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

int mutex = 1;

int full = 0;

int empty = 10, x = 0;

void producer()

{

--mutex;

++full;

--empty;

x++;

printf("\nProducer produces"

"item %d",

x);

++mutex;

}

void consumer()

{

--mutex;

--full;

++empty;

printf("\nConsumer consumes "

"item %d",

x);

x--;

++mutex;

}

int main()

{

int n, i;

printf("\n1. Press 1 for Producer"

"\n2. Press 2 for Consumer"

"\n3. Press 3 for Exit");

#pragma omp critical

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

switch (n) {

case 1:

if ((mutex == 1)

&& (empty != 0)) {

producer();

}

else {

printf("Buffer is full!");

}

break;

case 2:

if ((mutex == 1)

&& (full != 0)) {

consumer();

}

else {

printf("Buffer is empty!");

}

break;

case 3:

exit(0);

break;

}

}

}

**OUTPUT:**

1. Press 1 for Producer

2. Press 2 for Consumer

3. Press 3 for Exit

Enter your choice:1

Producer producesitem 1

Enter your choice:1

Producer producesitem 2

Enter your choice:2

Consumer consumes item 2

Enter your choice:3

# #PROGRAM 7

**AIM:**

To write a C program for implementation of FIFO page replacement algorithm.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the necessary variables.

Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen. Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults.

Step 10: Stop the program.

**PROGRAM:**

#include <stdio.h>

int main()

{

int referenceString[10], pageFaults = 0, m, n, s, pages, frames;

printf("\nEnter the number of Pages:\t");

scanf("%d", &pages);

printf("\nEnter reference string values:\n");for( m = 0; m < pages; m++)

{

printf("Value No. [%d]:\t", m + 1);

scanf("%d", &referenceString[m]);

}

printf("\n What are the total number of frames:\t");

{

scanf("%d", &frames);

}

int temp[frames];

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

{

temp[m] = -1;

}

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

{

s = 0;

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

{

if(referenceString[m] == temp[n])

{

s++;

pageFaults--;

}}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

{

temp[m] = referenceString[m];

}

else if(s == 0)

{

temp[(pageFaults - 1) % frames] = referenceString[m];

}

printf("\n");

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

{

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

}

}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}

**OUTPUT:**

Enter the number of Pages: 3

Enter reference string values:

Value No. [1]: 6

Value No. [2]: 4

Value No. [3]: 8

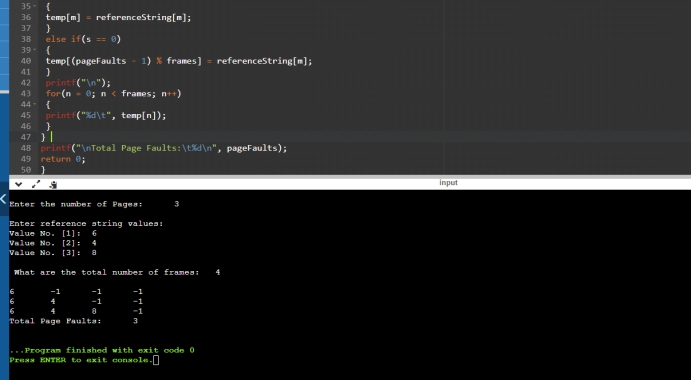
What are the total number of frames: 4

6 -1 -1 -1

6 4 -1 -1

6 4 8 -1

Total Page Faults: 3



**RESULT:**

Hence the FIFO page replacement has been executed successfully.

**#PROGRAM 8**

**AIM:**

To write a C program for implementation of LRU page replacement algorithm.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Declare the necessary variables.

Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen. Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults.

Step 10: Stop the program.

**PROGRAM:**

#include<stdio.h>

int findLRU(int time[], int n){

int i, minimum = time[0], pos = 0;

for(i = 1; i < n; ++i){

if(time[i] < minimum){

minimum = time[i];

pos = i;

}

}

return pos;

}

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2, i, j, pos, faults = 0;

printf("Enter number of frames: ");

scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == pages[i]){

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){if(frames[j] == -1){

counter++;

faults++;

frames[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

if(flag2 == 0){

pos = findLRU(time, no\_of\_frames);

counter++;

faults++;

frames[pos] = pages[i];

time[pos] = counter;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

printf("%d\t", frames[j]);

}

}

printf("\n\nTotal Page Faults = %d", faults);return 0;

}

**OUTPUT:**

Enter number of frames: 5

Enter number of pages: 4

Enter reference string: 3 4 7 9

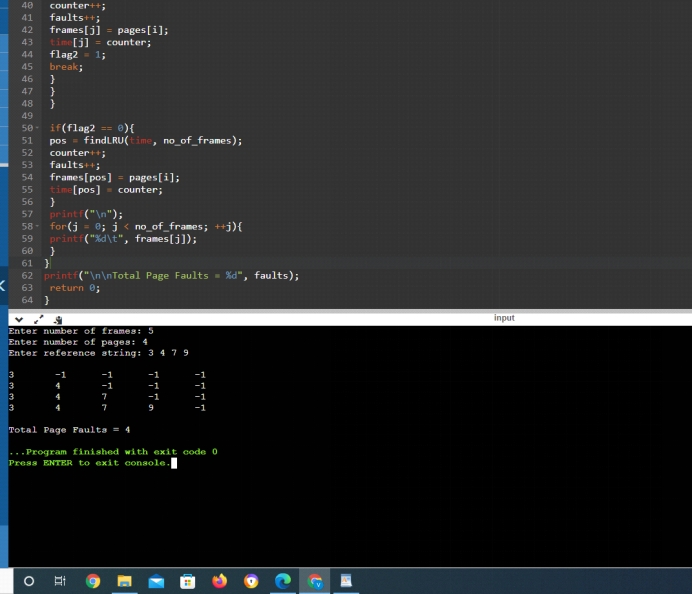
3 -1 -1 -1 -1

3 4 -1 -1 -1

3 4 7 -1 -1

3 4 7 9 -1

Total Page Faults = 4



**RESULT:**

Hence the program has been executed successfully

**#PROGRAM 9**

**AIM:**

To write a C program for implementation of OPTIMAL page replacement algorithm.

**ALGORITHM**:

Step 1: Start the program.

Step 2: Declare the necessary variables.

Step 3: Enter the number of frames.

Step 4: Enter the reference string end with zero.

Step 5: FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen. Step 6: When a page is brought into memory, it is inserted at the tail of the queue.

Step 7: Initially all the three frames are empty.

Step 8: The page fault range increases as the no of allocated frames also increases.

Step 9: Print the total number of page faults.

Step 10: Stop the program.

**PROGRAM:**

#include<stdio.h>

int main()

{

int no\_of\_frames, no\_of\_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k,

pos, max, faults = 0;

printf("Enter number of frames: ");scanf("%d", &no\_of\_frames);

printf("Enter number of pages: ");

scanf("%d", &no\_of\_pages);

printf("Enter page reference string: ");

for(i = 0; i < no\_of\_pages; ++i){

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

}

for(i = 0; i < no\_of\_frames; ++i){

frames[i] = -1;

}

for(i = 0; i < no\_of\_pages; ++i){

flag1 = flag2 = 0;

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == pages[i]){

flag1 = flag2 = 1;

break;

}

}

if(flag1 == 0){

for(j = 0; j < no\_of\_frames; ++j){

if(frames[j] == -1){

faults++;

frames[j] = pages[i];

flag2 = 1;break;

}

}

}

if(flag2 == 0){

flag3 =0;

for(j = 0; j < no\_of\_frames; ++j){

temp[j] = -1;

for(k = i + 1; k < no\_of\_pages; ++k){

if(frames[j] == pages[k]){

temp[j] = k;

break;

}

}

}

for(j = 0; j < no\_of\_frames; ++j){

if(temp[j] == -1){

pos = j;

flag3 = 1;

break;

}

}

if(flag3 ==0){

max = temp[0];pos = 0;

for(j = 1; j < no\_of\_frames; ++j){

if(temp[j] > max){

max = temp[j];

pos = j;

}

}

}

frames[pos] = pages[i];

faults++;

}

printf("\n");

for(j = 0; j < no\_of\_frames; ++j){

printf("%d\t", frames[j]);

}

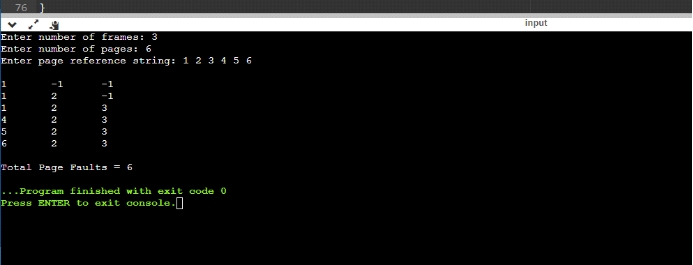
}

printf("\n\nTotal Page Faults = %d", faults);

return 0;

}

**OUTPUT:**



**RESULT:**

Hence the program has been executed successfully

**#program 10**

**Aim:**

Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a program to simulate the file allocation strategy.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of memory partition and their sizes.

Step 3: Get the number of processes and values of block size for each process.

Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.

Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates it.

Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.

Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.

Step 8: Stop the program

**Program:**

#include<stdio.h>

int main()

{

char name[10][30];

int start[10],length[10],num;

printf("Enter the number of files to be allocated\n");

scanf("%d",&num);

int count=0,k,j;

for(int i=0;i<num;i++)

{

printf("Enter the name of the file %d\n",i+1);

scanf("%s",&name[i][0]);

printf("Enter the start block of the file %d\n",i+1);

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

printf("Enter the length of the file %d\n",i+1);

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

for(j=0,k=1;j<num && k<num;j++,k++)

{

if(start[j+1]<=start[j] || start[j+1]>=length[j])

{

}

else

{

count++;

}

}

if(count==1)

{

printf("%s cannot be allocated disk space\n",name[i]);

}

}

printf("File Allocation Table\n");

printf("%s%40s%40s\n","File Name","Start Block","Length");

printf("%s%50d%50d\n",name[0],start[0],length[0]);

for(int i=0,j=1;i<num && j<num;i++,j++)

{

if(start[i+1]<=start[i] || start[i+1]>=length[i])

{

printf("%s%50d%50d\n",name[j],start[j],length[j]);

}

}

return 0;

}

**Output:**

Enter the number of files to be allocated

2

Enter the name of the file 1

hhl

Enter the start block of the file 1

13

Enter the length of the file 1

2

Enter the name of the file 2

pp

Enter the start block of the file 2

12

Enter the length of the file 2

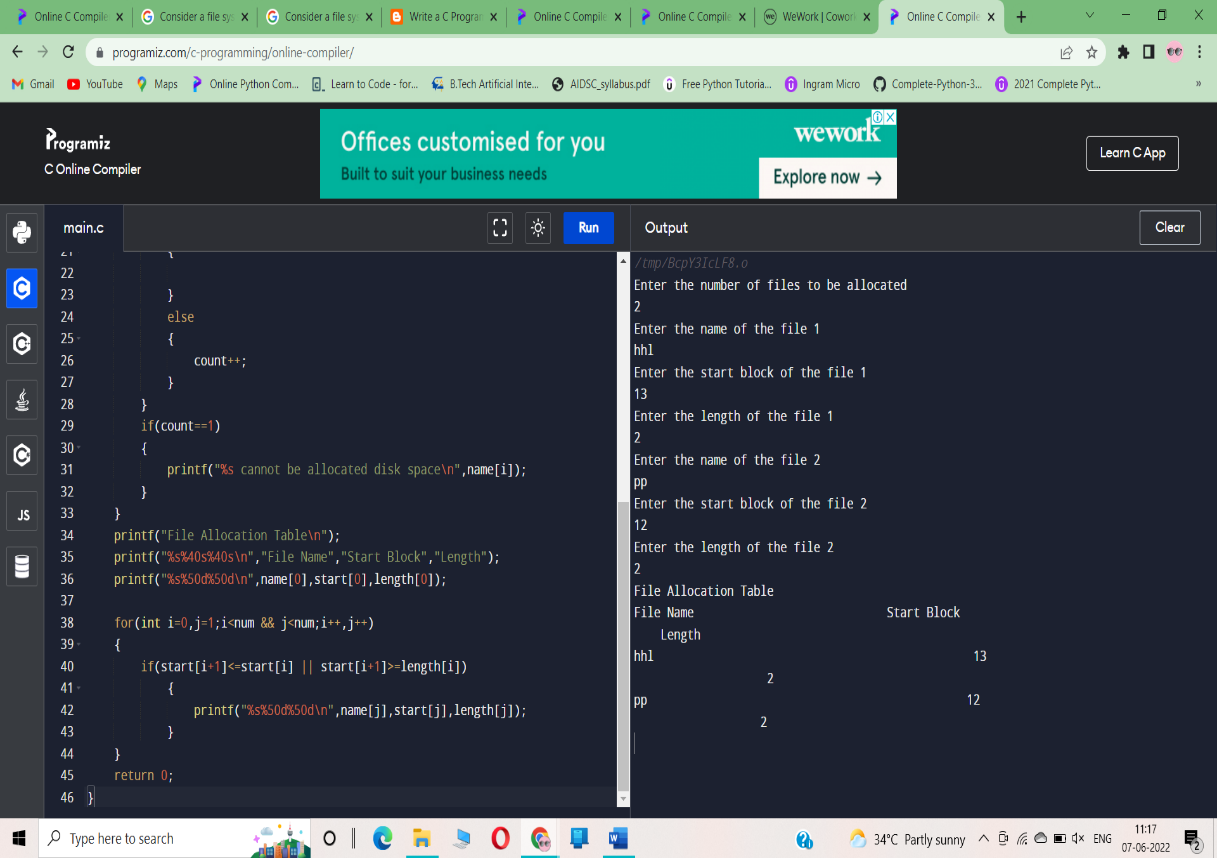
2

File Allocation Table

File Name Start Block Length

hhl 13 2

pp 12 2



**Result:**

Thus a C program to simulate the sequential file allocation strategy has been created successfully

**#program 11**

**Aim:**

Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a program to simulate the file allocation strategy

**ALGORITHM:**

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any consumer.If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process

**Program:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;

for(i=0;i<50;i++)

f[i]=0;

x:printf("Enter the index block: ");

scanf("%d",&ind);

if(f[ind]!=1)

{

printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);

scanf("%d",&n);

}

else

{

printf("%d index is already allocated \n",ind);

goto x;

}

y: count=0;

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

{

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

if(f[index[i]]==0)

count++;

}

if(count==n)

{

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

f[index[j]]=1;

printf("Allocated\n");

printf("File Indexed\n");

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

printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);

}

else

{

printf("File in the index is already allocated \n");

printf("Enter another file indexed");

goto y;

}

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

}

**Output:**

Enter the index block: 5

Enter no of blocks needed and no of files for the index 5 on the disk :

4

1 2 3 4

Allocated

File Indexed

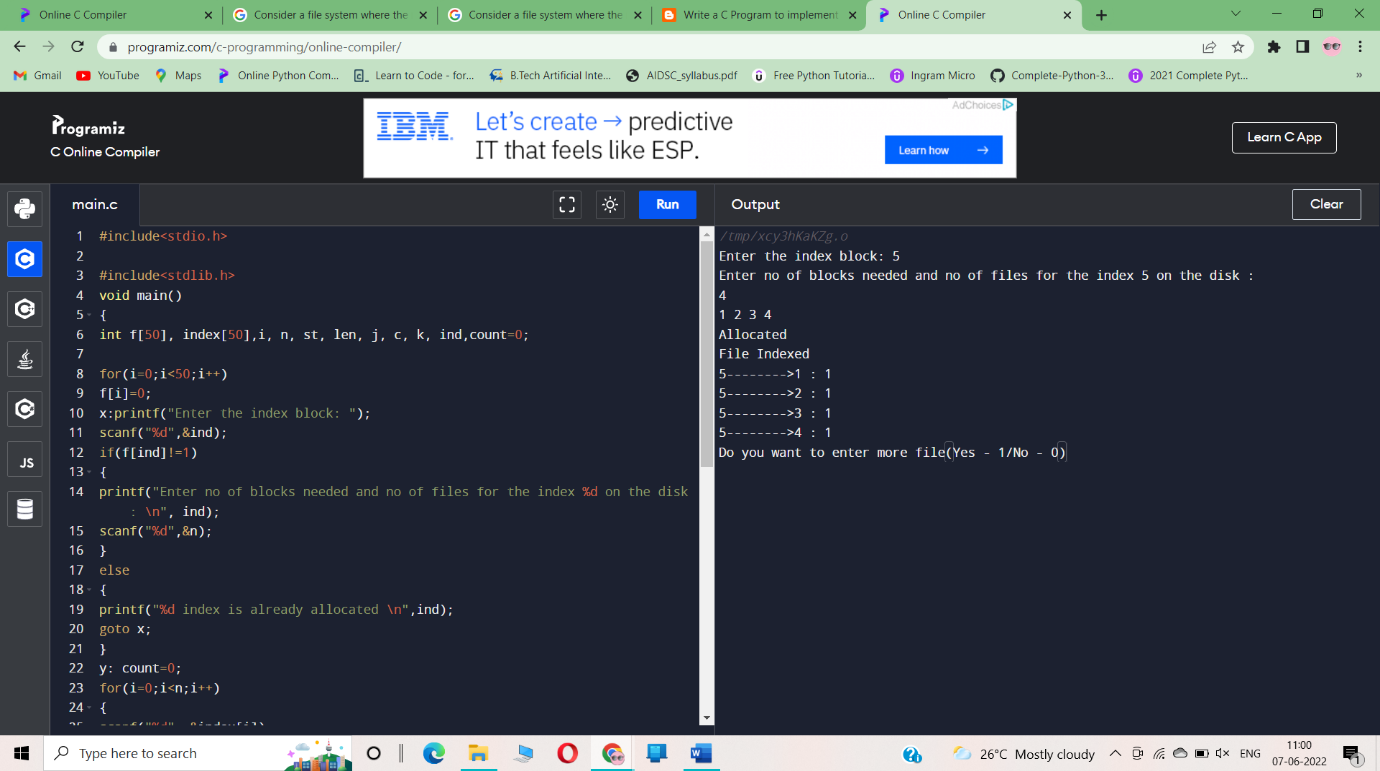
5-------->1 : 1

5-------->2 : 1

5-------->3 : 1

5-------->4 : 1

Do you want to enter more file(Yes - 1/No - 0)1



**Result:**

Thus a C program to simulate the index file allocation strategy has been created successfully

**#program12**

**Aim:**

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a program to simulate the file allocation strategy.

**ALGORITHM:**

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

Step 5: When the page fault occurs replace page present at the bottom of the stack

Step 6: Stop the allocation.

**Program:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

int f[50], p,i, st, len, j, c, k, a;

for(i=0;i<50;i++)

f[i]=0;

printf("Enter how many blocks already allocated: ");

scanf("%d",&p);

printf("Enter blocks already allocated: ");

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

{

scanf("%d",&a);

f[a]=1;

}

x: printf("Enter index starting block and length: ");

scanf("%d%d", &st,&len);

k=len;

if(f[st]==0)

{

for(j=st;j<(st+k);j++)

{

if(f[j]==0)

{

f[j]=1;

printf("%d-------->%d\n",j,f[j]);

}

else

{

printf("%d Block is already allocated \n",j);

k++;

}

}

}

else

printf("%d starting block is already allocated \n",st);

printf("Do you want to enter more file(Yes - 1/No - 0)");

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

}

**Output:**

Enter how many blocks already allocated: 3

Enter blocks already allocated: 1 35

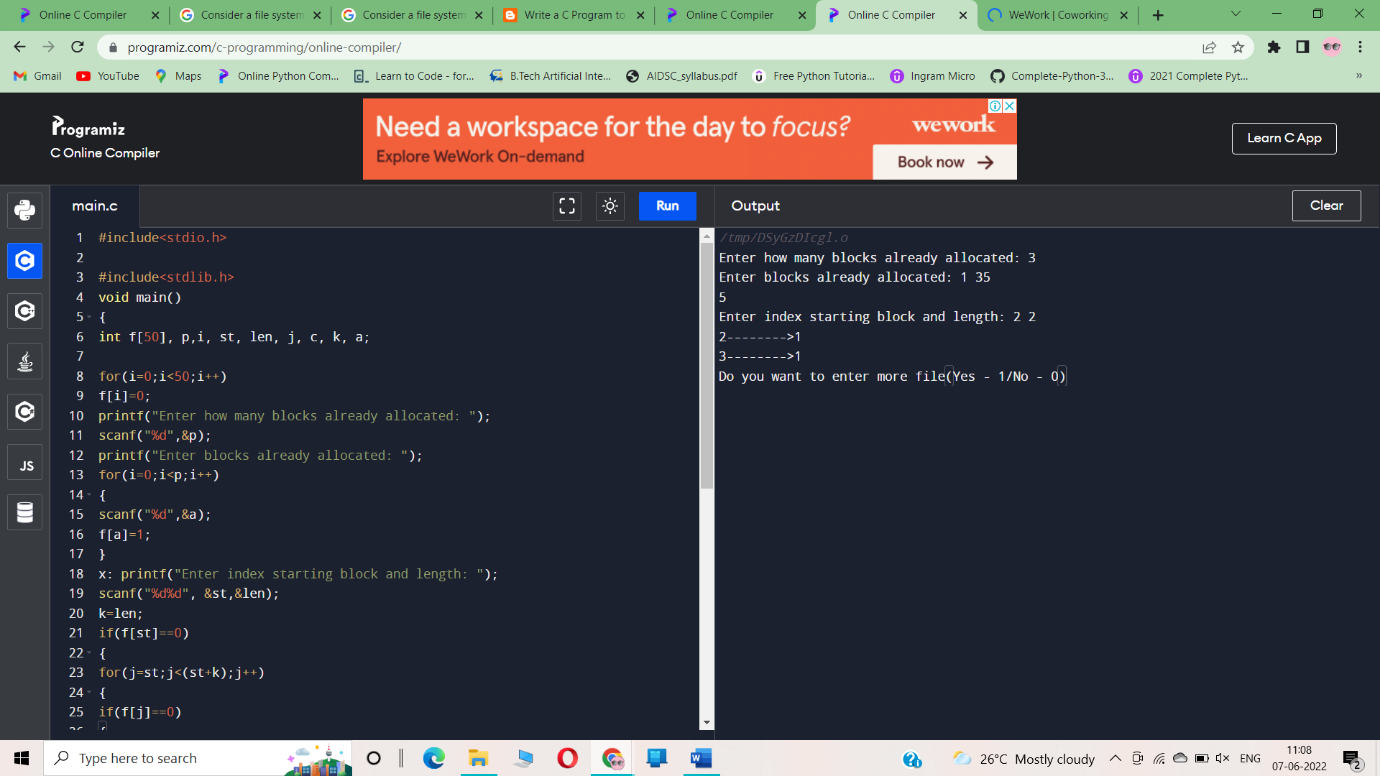
5

Enter index starting block and length: 2 2

2-------->1

3-------->1

Do you want to enter more file(Yes - 1/No - 0)



**Result:**

Thus a C program to simulate the linked file allocation strategy has been created successfully

**#program 13**

**Aim:**

Construct a C program to simulate the First Come First Served disk scheduling algorithm

**ALGORITHM:**  
1. Input the maximum number of cylinders and work queue and its head starting position.  
2. First Come First Serve Scheduling (FCFS) algorithm – The operations are performed in order requested.  
3. There is no reordering of work queue.  
4. Every request is serviced, so there is no starvation.  
5. The seek time is calculated.  
6. Shortest Seek Time First Scheduling (SSTF) algorithm – This algorithm selects the request with the minimum seek time from the current head position.  
7. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.  
8. The seek time is calculated.  
9. SCAN Scheduling algorithm – The disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.  
10. At the other end, the direction of head movement is reversed, and servicing continues.  
11. The head continuously scans back and forth across the disk.  
12. The seek time is calculated.  
13. Display the seek time and terminate the program

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial;

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);

// logic for FCFS disk scheduling

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

{

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

initial=RQ[i];

}

printf("Total head moment is %d",TotalHeadMoment);

return 0;

}

**OUTPUT:**

Enter the number of Requests

4

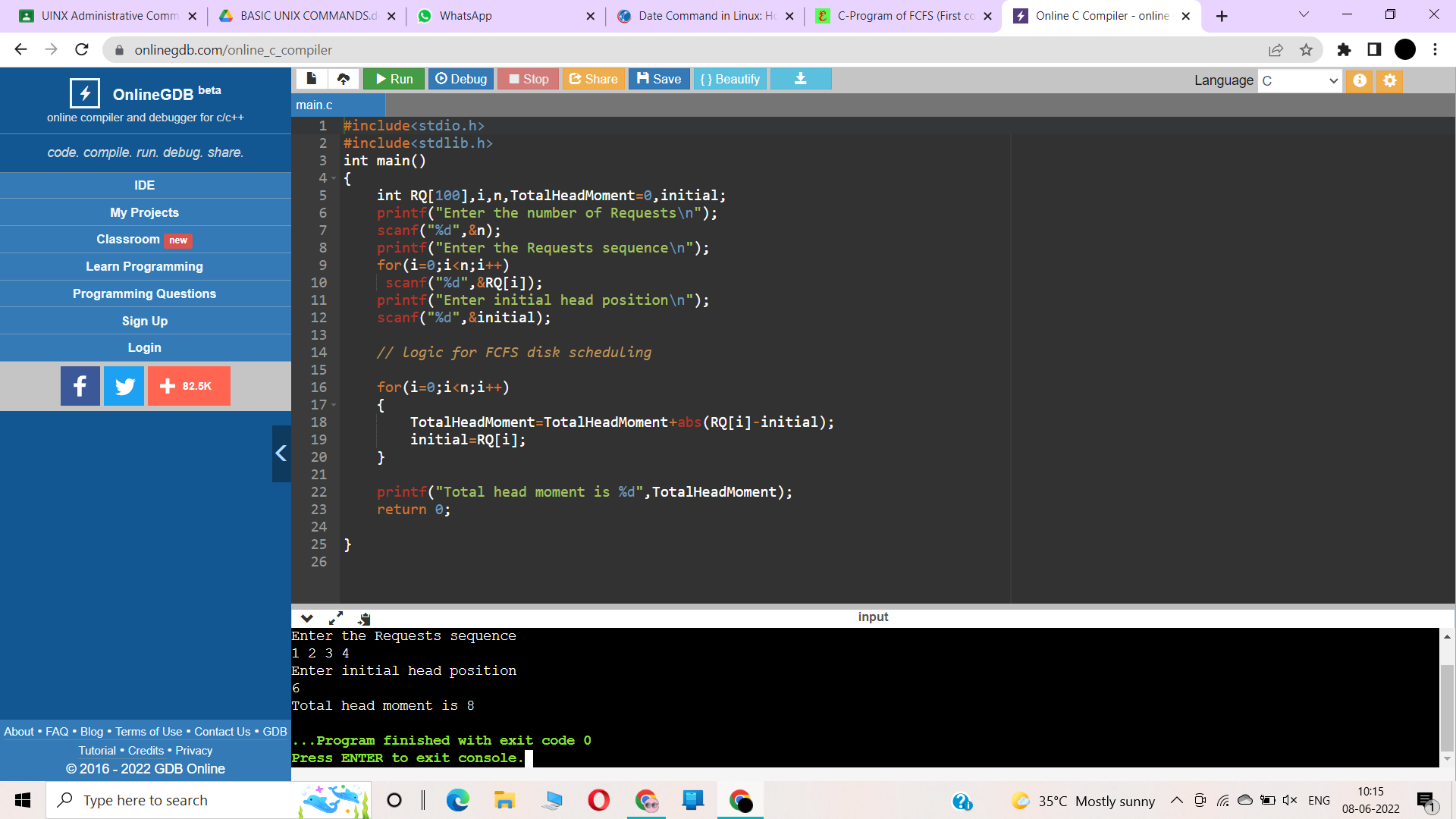
Enter the Requests sequence

1 2 3 4

Enter initial head position

55 6

Total head moment is 8



**RESULT:**

Thus a C program to simulate the First Come First Served disk scheduling algorithm

**#program 14**

**Aim:**

Illustrate the various File Access Permission and different types users in Linux

**ALGORITHM:**

1.Random access file in C enables us to read or write any data in our disk file without reading or writing every piece of data before it.

2.ftell() is used to find the position of the file pointer from the starting of the file.

3.rewind() is used to move the file pointer to the beginning of the file.

4.The fseek() function is used to move the file position to a desired location.

5.fseek() function can be used to find a specific record in a file provided we already know where the record starts in the file and its size.

6.Reading and writing to files are accomplished by combining the single letters "r", "b", "w", "a", and "+" with the other letters to form one or more file mode specifiers.

7.File Mode combinations allow us to accomplish reading and writing operations simultaneously.

**PROGRAM:**

#include<stdio.h>

int main()

{

FILE \*fp;

fp = fopen("D:\\DEVC++\\file1.txt","r");

if(!fp)

{

printf("Error in opening file\n");

return 0;

}

//Initially the file pointer points to the starting of the file.

printf("Position of the pointer : %ld\n",ftell(fp));

char ch;

while(fread(&ch,sizeof(ch),1,fp)==1)

{

//Here we traverse the entire file and print it's contents until we reach it's end.

printf("%c",ch);

}

printf("\nPosition of the pointer : %ld\n",ftell(fp));

//Below rewind() is going to bring it back to it's original position.

rewind(fp);

printf("\n USING REWIND Position of the pointer : %ld\n",ftell(fp));

printf("\nUSING FSEEK.....");

fseek(fp, 6, 0);

while(fread(&ch,sizeof(ch),1,fp)==1)

{

printf("%c",ch);

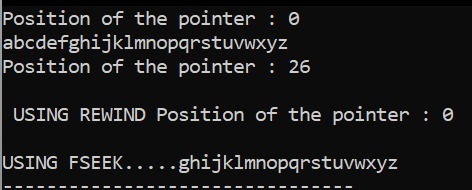
}

fclose(fp);

return 0;

}

**OUTPUT:**



**Result:**

Thus, the program for file access permission by using different users has been executed successfully

**#program 15**

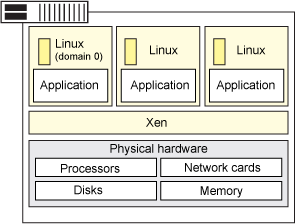
**Xen:**

Xen is a type 1 hypervisor that creates logical pools of system resources so that many virtual machines can share the same physical resources.

Xen is a hypervisor that runs directly on the system hardware. Xen inserts a virtualization layer between the system hardware and the virtual machines, turning the system hardware into a pool of logical computing resources that Xen can dynamically allocate to any guest operating system. The operating systems running in virtual machines interact with the virtual resources as if they were physical resources.

Figure 1 shows a system with Xen running virtual machines.

##### Figure 1. The Xen architecture



Xen is running three virtual machines. Each virtual machine is running a guest operating system and applications independent of other virtual machines while sharing the same physical resources.

## **Features**

The following are key concepts of the Xen architecture:

* Full virtualization.
* Xen can run multiple guest OS, each in its on VM.
* Instead of a driver, lots of great stuff happens in the Xen daemon, xend.

### **Full virtualization**

Most hypervisors are based on full virtualization which means that they completely emulate all hardware devices to the virtual machines. Guest operating systems do not require any modification and behave as if they each have exclusive access to the entire system.

Full virtualization often includes performance drawbacks because complete emulation usually demands more processing resources (and more overhead) from the hypervisor. Xen is based on paravirtualization; it requires that the guest operating systems be modified to support the Xen operating environment. However, the user space applications and libraries do not require modification.

Operating system modifications are necessary for reasons like:

* So that Xen can replace the operating system as the most privileged software.
* So that Xen can use more efficient interfaces (such as virtual block devices and virtual network interfaces) to emulate devices — this increases performance.

### **Xen can run multiple guest OS each in its on VM**

Xen can run several guest operating systems each running in its own virtual machine or domain. When Xen is first installed, it automatically creates the first domain, Domain 0 (or dom0).

Domain 0 is the management domain and is responsible for managing the system. It performs tasks like building additional domains (or virtual machines), managing the virtual devices for each virtual machine, suspending virtual machines, resuming virtual machines, and migrating virtual machines. Domain 0 runs a guest operating system and is responsible for the hardware devices.

### **Instead of a driver, lots of great stuff happens in the Xen daemon**

The Xen daemon, xend, is a Python program that runs in dom0. It is the central point of control for managing virtual resources across all the virtual machines running on the Xen hypervisor. Most of the command parsing, validation, and sequencing happens in user space in xend and not in a driver.

IBM supports the SUSE Linux Enterprise Edition (SLES) 10 version of Xen which supports the following configuration:

* Four virtual machines per processor and up to 64 virtual machines per physical system.
* SLES 10 guest operating systems (paravirtualized only).

## **Deploying virtualization**

To deploy virtualization for Xen:

* Install Xen on the system.
* Create and configure virtual machines (this includes the guest operating system).

Install the Xen software using one of the following methods:

* **Interactive install:** Use this procedure to install directly on dedicated virtual machine on the Xen server. This dedicated virtual machine is referred to as the client computer in the install procedure.
* **Install from CommCell console:** Use this procedure to install remotely on a dedicated virtual machine on the Xen server.

## **Managing your virtual machines**

There are several virtual machine managers available including:

* **Open source mangers:** OpenXenManager, an open source clone of Citrix's XenServer XenCenter and manages both XCP and Citrix's XenServer. Xen Cloud Control System (XCCS) is a lightweight front end package for the excellent Xen Cloud Platform cloud computing system. Zentific, a web-based management interface for the effective control of virtual machines running upon the Xen hypervisor.
* **Commercial managers:** Convirture: ConVirt is a centralized management solution that lets you provision, monitor, and manage the complete life cycle of your Xen deployment. Citrix XenCenter is a Windows-native graphical user interface for managing Citrix XenServer and XCP. Versiera is a web-based Internet technology designed to securely manage and monitor both cloud environments and enterprises with support for Linux, FreeBSD, OpenBSD, NetBSD, OS X, Windows, Solaris, OpenWRT, and DD-WRT.

## **Choosing Xen**

On the pro side:

* The Xen server is built on the open source Xen hypervisor and uses a combination of paravirtualization and hardware-assisted virtualization. This collaboration between the OS and the virtualization platform enables the development of a simpler hypervisor that delivers highly optimized performance.
* Xen provides sophisticated workload balancing that captures CPU, memory, disk I/O, and network I/O data; it offers two optimization modes: one for performance and another for density.
* The Xen server takes advantage of a unique storage integration feature called the Citrix Storage Link. With it, the sysadmin can directly leverage features of arrays from such companies as HP, Dell Equal Logic, NetApp, EMC, and others.
* The Xen server includes multicore processor support, live migration, physical-server-to-virtual-machine conversion (P2V) and virtual-to-virtual conversion (V2V) tools, centralized multiserver management, real-time performance monitoring, and speedy performance for Windows and Linux.

On the con side:

* Xen has a relatively large footprint and relies on Linux in dom0.
* Xen relies on third-party solutions for hardware device drivers, storage, backup and recovery, and fault tolerance.
* Xen gets bogged down with anything with a high I/O rate or anything that sucks up resources and starves other VMs.
* Xen's integration can be problematic; it could become a burden on your Linux kernel over time.
* XenServer 5 is missing 802.1Q virtual local area network (VLAN) trunking; as for security, it doesn't offer directory services integration, role-based access controls, or security logging and auditing or administrative actions.

v

# **VMware:**

VMware is a virtualization and cloud computing software provider based in Palo Alto, Calif. Founded in 1998, VMware is a subsidiary of Dell Technologies. EMC Corporation originally acquired VMware in 2004;

**#program 16**

Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

**PROGRAM:**

#include<stdio.h>

int main()

{

int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg\_wt,avg\_tat;

printf("Enter Total Number of Process:");

scanf("%d",&n);

printf("\nEnter Burst Time and Priority\n");

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

{

printf("\nP[%d]\n",i+1);

printf("Burst Time:");

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

printf("Priority:");

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

p[i]=i+1;

}

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

{

pos=i;

for(j=i+1;j<n;j++)

{

if(pr[j]<pr[pos])

pos=j;

}

temp=pr[i];

pr[i]=pr[pos];

pr[pos]=temp;

temp=bt[i];

bt[i]=bt[pos];

bt[pos]=temp;

temp=p[i];

p[i]=p[pos];

p[pos]=temp;

}

wt[0]=0;

for(i=1;i<n;i++)

{

wt[i]=0;

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

wt[i]+=bt[j];

total+=wt[i];

}

avg\_wt=total/n;

total=0;

printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

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

{

tat[i]=bt[i]+wt[i];

total+=tat[i];

printf("\nP[%d]\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);

}

avg\_tat=total/n;

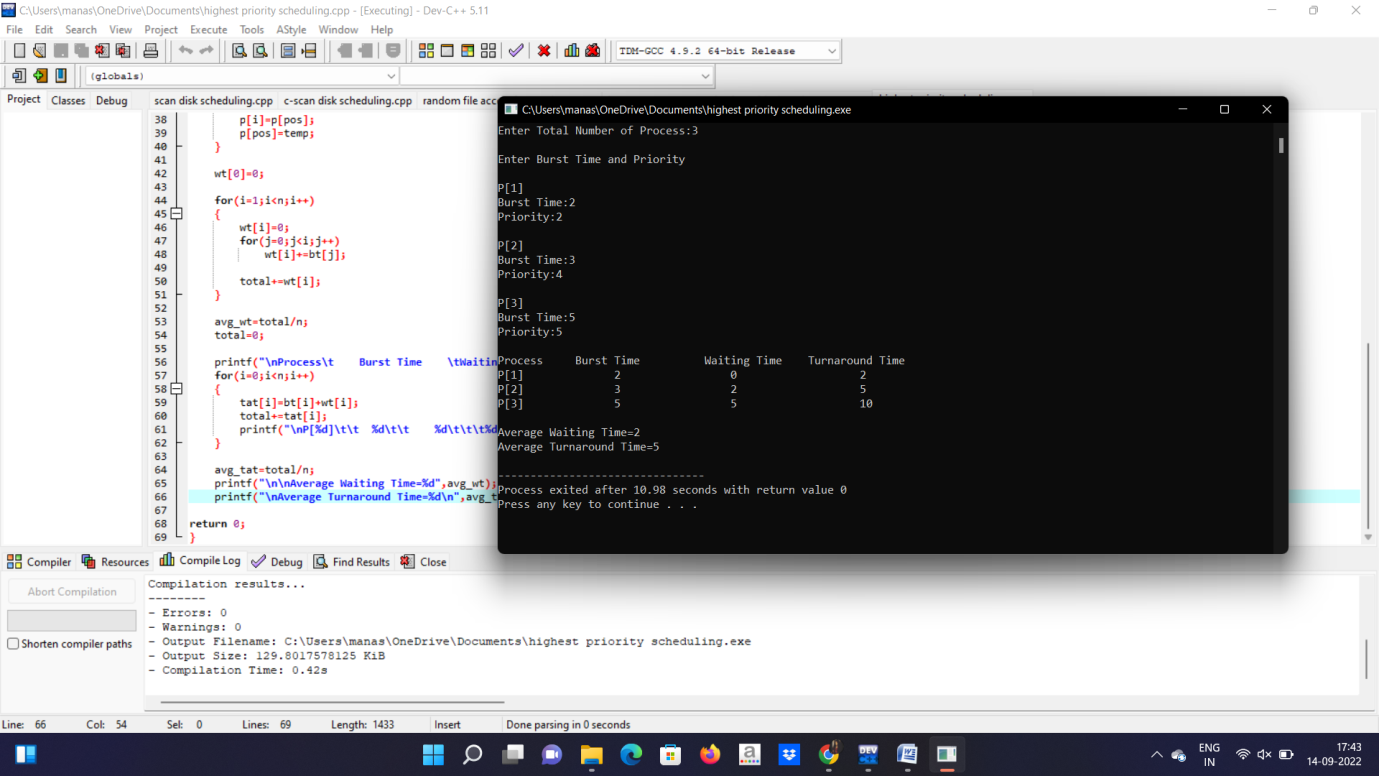
printf("\n\nAverage Waiting Time=%d",avg\_wt);

printf("\nAverage Turnaround Time=%d\n",avg\_tat);

return 0;

}

**OUTPUT:**



**#program 17**

. Construct a C program to simulate Round Robin scheduling algorithm with C.

**PROGRAM:**

**#include<stdio.h>**

**int main()**

**{**

**int count,j,n,time,remain,flag=0,time\_quantum;**

**int wait\_time=0,turnaround\_time=0,at[10],bt[10],rt[10];**

**printf("Enter Total Process:\t ");**

**scanf("%d",&n);**

**remain=n;**

**for(count=0;count<n;count++)**

**{**

**printf("Enter Arrival Time and Burst Time for Process Process Number %d :",count+1);**

**scanf("%d",&at[count]);**

**scanf("%d",&bt[count]);**

**rt[count]=bt[count];**

**}**

**printf("Enter Time Quantum:\t");**

**scanf("%d",&time\_quantum);**

**printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");**

**for(time=0,count=0;remain!=0;)**

**{**

**if(rt[count]<=time\_quantum && rt[count]>0)**

**{**

**time+=rt[count];**

**rt[count]=0;**

**flag=1;**

**}**

**else if(rt[count]>0)**

**{**

**rt[count]-=time\_quantum;**

**time+=time\_quantum;**

**}**

**if(rt[count]==0 && flag==1)**

**{**

**remain--;**

**printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);**

**wait\_time+=time-at[count]-bt[count];**

**turnaround\_time+=time-at[count];**

**flag=0;**

**}**

**if(count==n-1)**

**count=0;**

**else if(at[count+1]<=time)**

**count++;**

**else**

**count=0;**

**}**

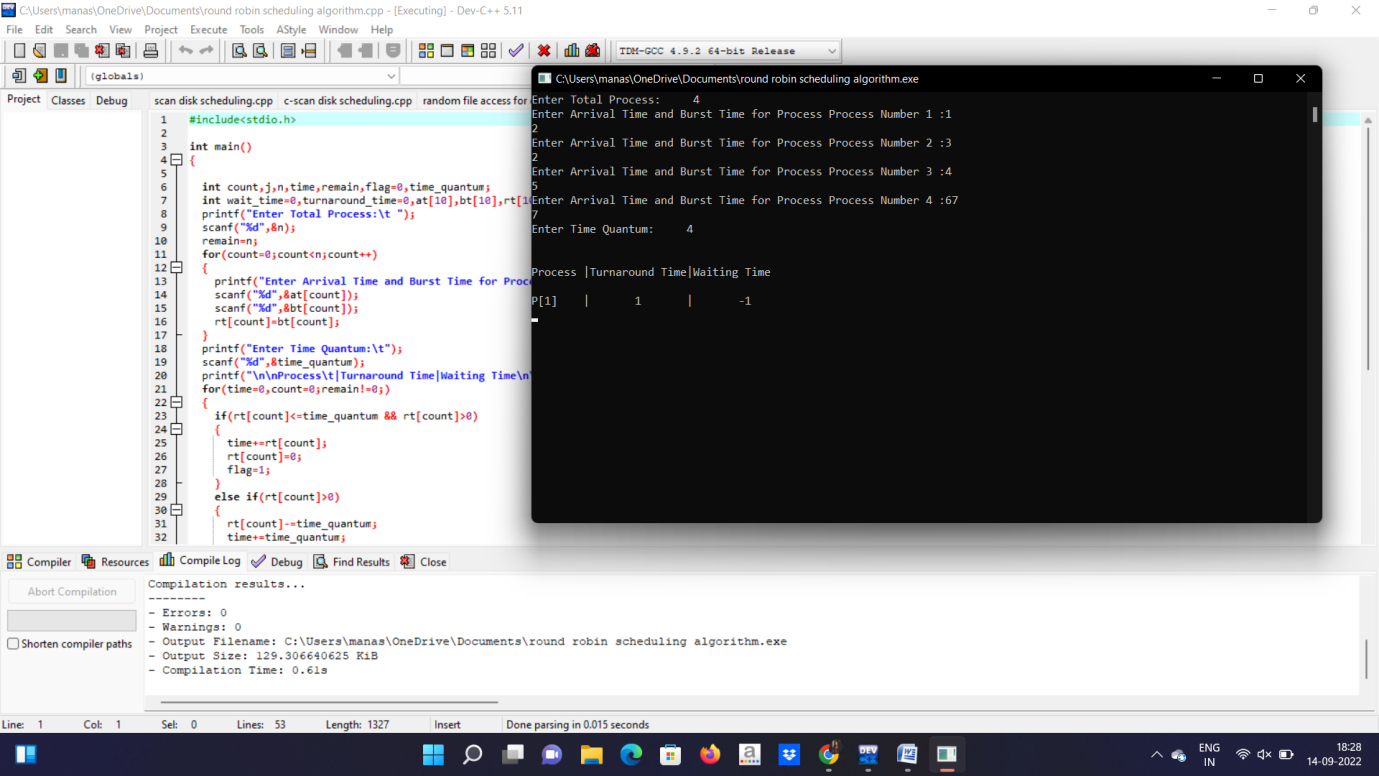
**printf("\nAverage Waiting Time= %f\n",wait\_time\*1.0/n);**

**printf("Avg Turnaround Time = %f",turnaround\_time\*1.0/n);**

**return 0;**

**}**

**OUTPUT:**



**#program 18**

Illustrate the concept of inter-process communication using shared memory with a C program.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char\*\* argv)

{

int id = 0, err = 0;

int\* mem;

if (id == -1)

perror("shmget");

else

printf("Allocated with shm id %d\n", (int)id);

perror("shmat");

\*mem = 100;

printf("Start other process-->");

getchar();

printf("mem is now %d\n", \*mem);

if (err == -1)

perror("shmctl");

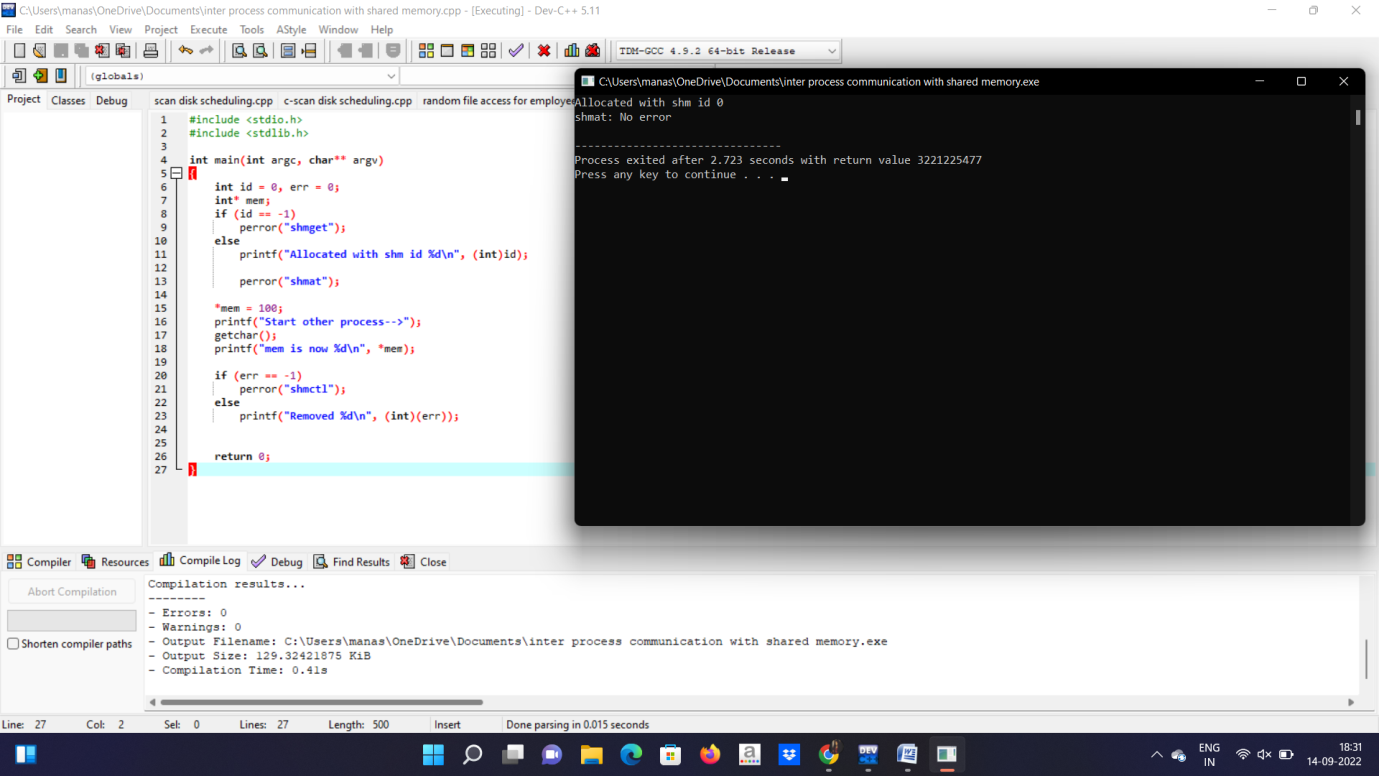
else

printf("Removed %d\n", (int)(err));

return 0;

}

**OUTPUT:**



**#program 19**

Illustrate the concept of multithreading using a C program.

**PROGRAM:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <unistd.h>**

**int sum = 0;**

**sem\_t mutex;**

**void \*add(void \*arg){**

**int \*ptr = (int \*) arg;**

**while(\*ptr != -1){**

**sem\_wait(&mutex);**

**sum += \*ptr;**

**printf("value: %d sum %d\n", \*ptr,sum );**

**sem\_post(&mutex);**

**ptr++;**

**}**

**return NULL;**

**}**

**int main(int argc, char \*args[]){**

**int A[4] = {1,2,3, -1};**

**int B[4] = {4,2,6, -1};**

**pthread\_t t\_a, t\_b;**

**sem\_init(&mutex, 0, 1);**

**pthread\_create(&t\_a , NULL, add, A);**

**pthread\_create(&t\_b, NULL, add, B);**

**pthread\_join(t\_a, NULL);**

**pthread\_join(t\_b, NULL);**

**printf("Total: %d\n", sum);**

**return 0;**

**}**

**OUTPUT:**

****

**#program 20**

Design a C program to simulate the concept of Dining-Philosophers problem

**PROGRAM:**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<pthread.h>**

**#include<semaphore.h>**

**#include<unistd.h>**

**sem\_t room;**

**sem\_t chopstick[5];**

**void \* philosopher(void \*);**

**void eat(int);**

**int main()**

**{**

**int i,a[5];**

**pthread\_t tid[5];**

**sem\_init(&room,0,4);**

**for(i=0;i<5;i++)**

**sem\_init(&chopstick[i],0,1);**

**for(i=0;i<5;i++){**

**a[i]=i;**

**pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]);**

**}**

**for(i=0;i<5;i++)**

**pthread\_join(tid[i],NULL);**

**}**

**void \* philosopher(void \* num)**

**{**

**int phil=\*(int \*)num;**

**sem\_wait(&room);**

**printf("\nPhilosopher %d has entered room",phil);**

**sem\_wait(&chopstick[phil]);**

**sem\_wait(&chopstick[(phil+1)%5]);**

**eat(phil);**

**sleep(2);**

**printf("\nPhilosopher %d has finished eating",phil);**

**sem\_post(&chopstick[(phil+1)%5]);**

**sem\_post(&chopstick[phil]);**

**sem\_post(&room);**

**}**

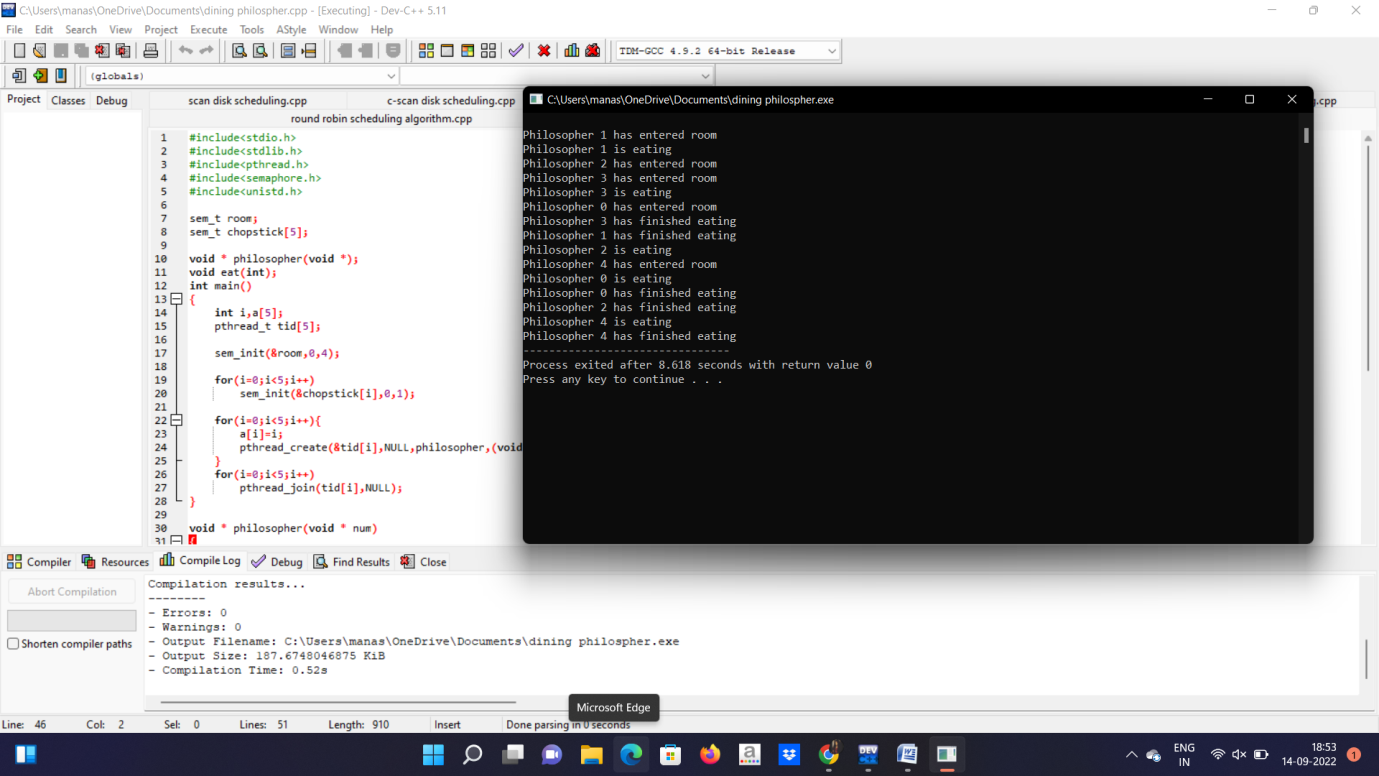
**void eat(int phil)**

**{**

**printf("\nPhilosopher %d is eating",phil);**

**}**

**OUTPUT:**

****

**#program 21**

Construct a C program for implementation of memory allocation using first fit strategy.

**PROGRAM:**

**#include<stdio.h>**

**void firstFit(int blockSize[], int m, int processSize[], int n)**

**{**

**int i, j;**

**int allocation[n];**

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

**{**

**allocation[i] = -1;**

**}**

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

**{**

**for (j = 0; j < m; j++)**

**{**

**if (blockSize[j] >= processSize[i])**

**{**

**allocation[i] = j;**

**blockSize[j] -= processSize[i];**

**break;**

**}**

**}**

**}**

**printf("\nProcess No.\tProcess Size\tBlock no.\n");**

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

**{**

**printf(" %i\t\t\t", i+1);**

**printf("%i\t\t\t\t", processSize[i]);**

**if (allocation[i] != -1)**

**printf("%i", allocation[i] + 1);**

**else**

**printf("Not Allocated");**

**printf("\n");**

**}**

**}**

**int main()**

**{**

**int m;**

**int n;**

**int blockSize[] = {100, 500, 200, 300, 600};**

**int processSize[] = {212, 417, 112, 426};**

**m = sizeof(blockSize) / sizeof(blockSize[0]);**

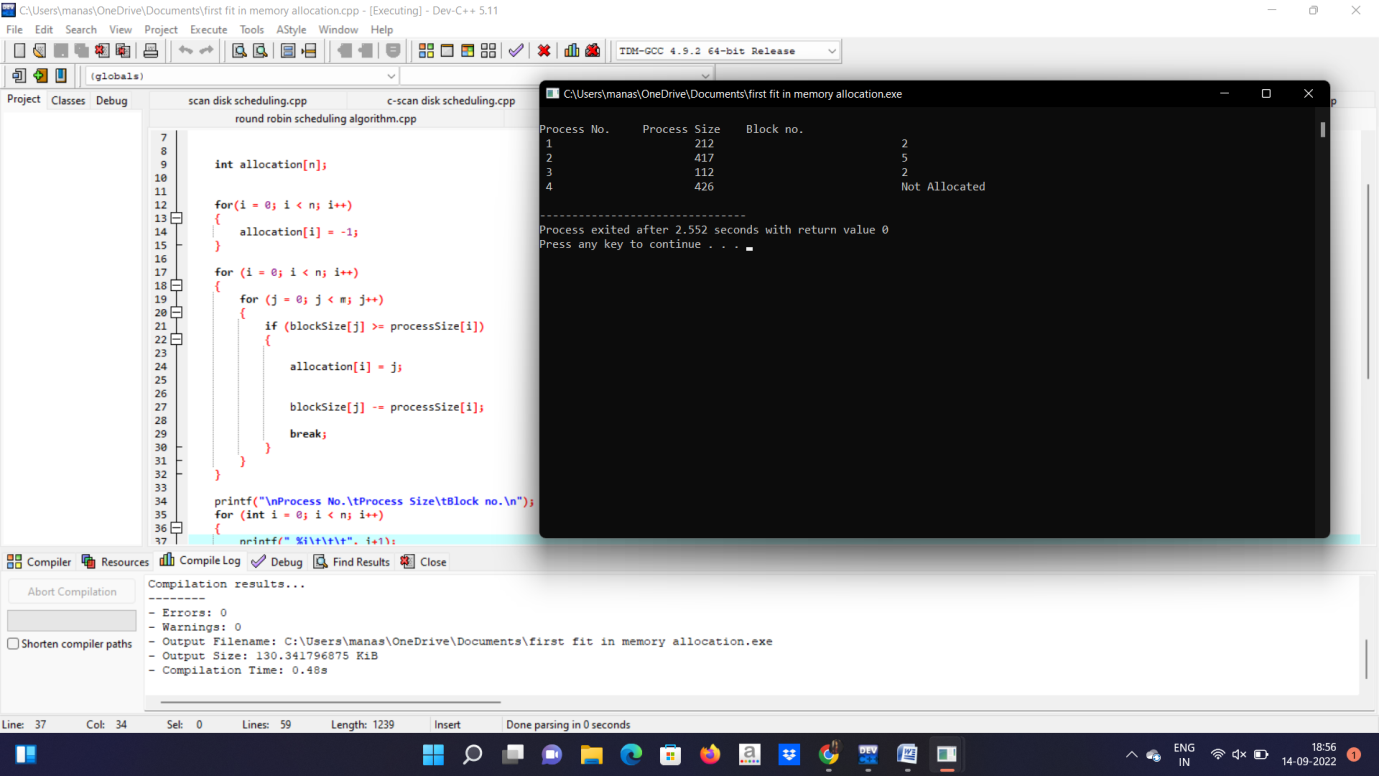
**n = sizeof(processSize) / sizeof(processSize[0]);**

**firstFit(blockSize, m, processSize, n);**

**return 0 ;**

**}**

**OUTPUT:**



**#program 22**

Construct a C program to organize the file using single level directory.

**PROGRAM:**

**#include<stdlib.h>**

**#include<string.h>**

**#include<stdio.h>**

**struct**

**{**

**char dname[10],fname[10][10];**

**int fcnt;**

**}dir;**

**int main()**

**{**

**int i,ch;**

**char f[30];**

**dir.fcnt = 0;**

**printf("\nEnter name of directory -- ");**

**scanf("%s", dir.dname);**

**while(1)**

**{**

**printf("\n\n1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5. Exit\nEnter your choice -- ");**

**scanf("%d",&ch);**

**switch(ch)**

**{**

**case 1: printf("\nEnter the name of the file -- ");**

**scanf("%s",dir.fname[dir.fcnt]);**

**dir.fcnt++;**

**break;**

**case 2: printf("\nEnter the name of the file -- ");**

**scanf("%s",f);**

**for(i=0;i<dir.fcnt;i++)**

**{**

**if(strcmp(f, dir.fname[i])==0)**

**{**

**printf("File %s is deleted ",f);**

**strcpy(dir.fname[i],dir.fname[dir.fcnt-1]); break; } }**

**if(i==dir.fcnt) printf("File %s not found",f);**

**else**

**dir.fcnt--;**

**break;**

**case 3: printf("\nEnter the name of the file -- ");**

**scanf("%s",f);**

**for(i=0;i<dir.fcnt;i++)**

**{**

**if(strcmp(f, dir.fname[i])==0)**

**{**

**printf("File %s is found ", f);**

**break;**

**}**

**}**

**if(i==dir.fcnt)**

**printf("File %s not found",f);**

**break;**

**case 4: if(dir.fcnt==0)**

**printf("\nDirectory Empty");**

**else**

**{**

**printf("\nThe Files are -- ");**

**for(i=0;i<dir.fcnt;i++)**

**printf("\t%s",dir.fname[i]);**

**}**

**break;**

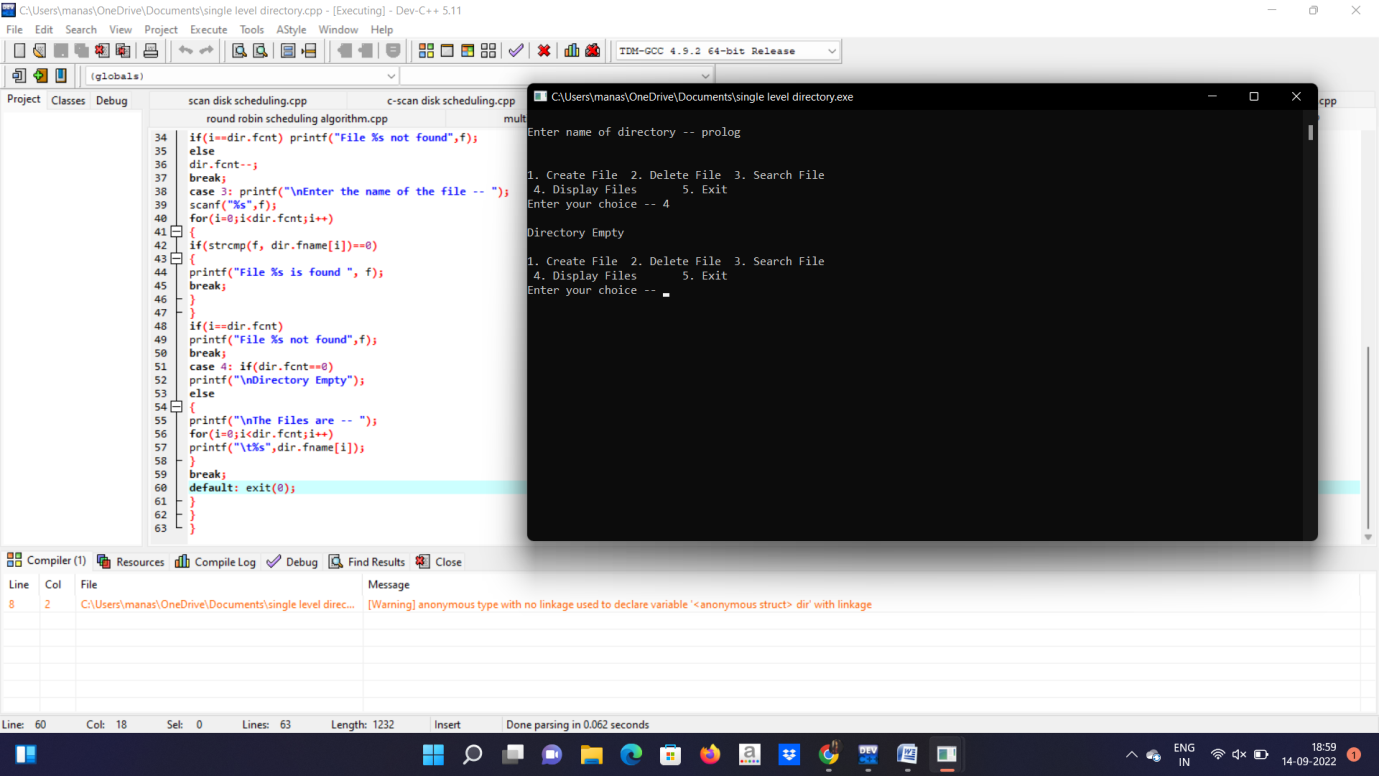
**default: exit(0);**

**}**

**}**

**}**

**OUTPUT:**

****

**#program 23**

Design a C program to organize the file using two level directory structure.

**PROGRAM:**

**#include<string.h>**

**#include<stdlib.h>**

**#include<stdio.h>**

**struct**

**{**

**char dname[10],fname[10][10];**

**int fcnt;**

**}dir[10];**

**int main()**

**{**

**int i,ch,dcnt,k;**

**char f[30], d[30];**

**dcnt=0;**

**while(1)**

**{**

**printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");**

**printf("\n4. Search File\t\t5. Display\t6. Exit\tEnter your choice -- ");**

**scanf("%d",&ch);**

**switch(ch)**

**{**

**case 1: printf("\nEnter name of directory -- ");**

**scanf("%s", dir[dcnt].dname);**

**dir[dcnt].fcnt=0;**

**dcnt++;**

**printf("Directory created");**

**break;**

**case 2: printf("\nEnter name of the directory -- ");**

**scanf("%s",d);**

**for(i=0;i<dcnt;i++)**

**if(strcmp(d,dir[i].dname)==0)**

**{**

**printf("Enter name of the file -- ");**

**scanf("%s",dir[i].fname[dir[i].fcnt]);**

**printf("File created");**

**break;**

**}**

**if(i==dcnt)**

**printf("Directory %s not found",d);**

**break;**

**case 3: printf("\nEnter name of the directory -- ");**

**scanf("%s",d);**

**for(i=0;i<dcnt;i++)**

**{**

**if(strcmp(d,dir[i].dname)==0)**

**{**

**printf("Enter name of the file -- ");**

**scanf("%s",f);**

**for(k=0;k<dir[i].fcnt;k++)**

**{**

**if(strcmp(f, dir[i].fname[k])==0)**

**{**

**printf("File %s is deleted ",f);**

**dir[i].fcnt--;**

**strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);**

**goto jmp;**

**}**

**}**

**printf("File %s not found",f);**

**goto jmp;**

**}**

**}**

**printf("Directory %s not found",d);**

**jmp : break;**

**case 4: printf("\nEnter name of the directory -- ");**

**scanf("%s",d);**

**for(i=0;i<dcnt;i++)**

**{**

**if(strcmp(d,dir[i].dname)==0)**

**{**

**printf("Enter the name of the file -- ");**

**scanf("%s",f);**

**for(k=0;k<dir[i].fcnt;k++)**

**{**

**if(strcmp(f, dir[i].fname[k])==0)**

**{**

**printf("File %s is found ",f);**

**goto jmp1;**

**}**

**}**

**printf("File %s not found",f);**

**goto jmp1;**

**}**

**}**

**printf("Directory %s not found",d);**

**jmp1: break;**

**case 5: if(dcnt==0)**

**printf("\nNo Directory's ");**

**else**

**{**

**printf("\nDirectory\tFiles");**

**for(i=0;i<dcnt;i++)**

**{**

**printf("\n%s\t\t",dir[i].dname);**

**for(k=0;k<dir[i].fcnt;k++)**

**printf("\t%s",dir[i].fname[k]);**

**}**

**}**

**break;**

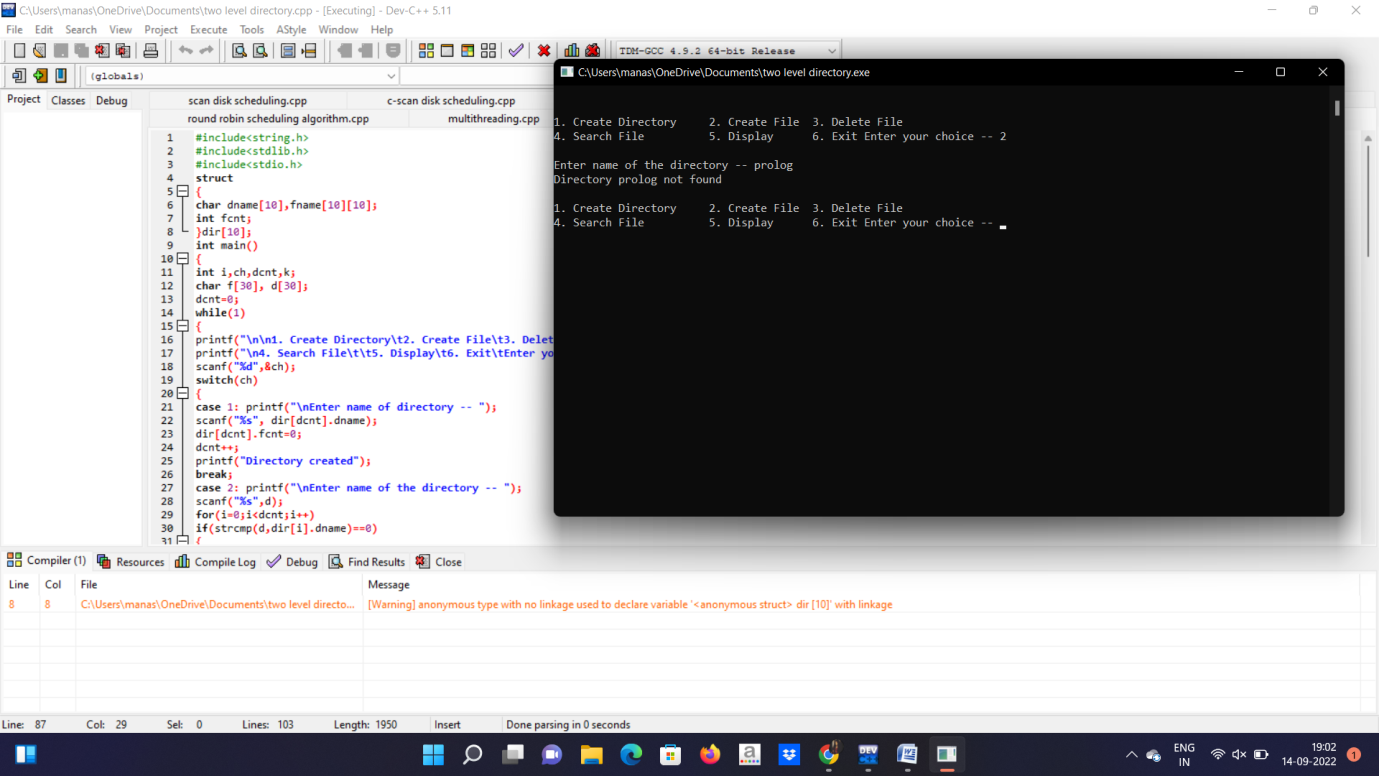
**default:exit(0);**

**}**

**}**

**}**

**OUTPUT:**

****

**#program24**

Develop a C program for implementing random access file for processing the employee details

**PROGRAM:**

**#include <stdio.h>**

**struct employee{**

**char name[30];**

**int empId;**

**float salary;**

**};**

**int main()**

**{**

**struct employee emp;**

**printf("\nEnter details :\n");**

**printf("Name ?:"); gets(emp.name);**

**printf("ID ?:"); scanf("%d",&emp.empId);**

**printf("Salary ?:"); scanf("%f",&emp.salary);**

**printf("\nEntered detail is:");**

**printf("Name: %s" ,emp.name);**

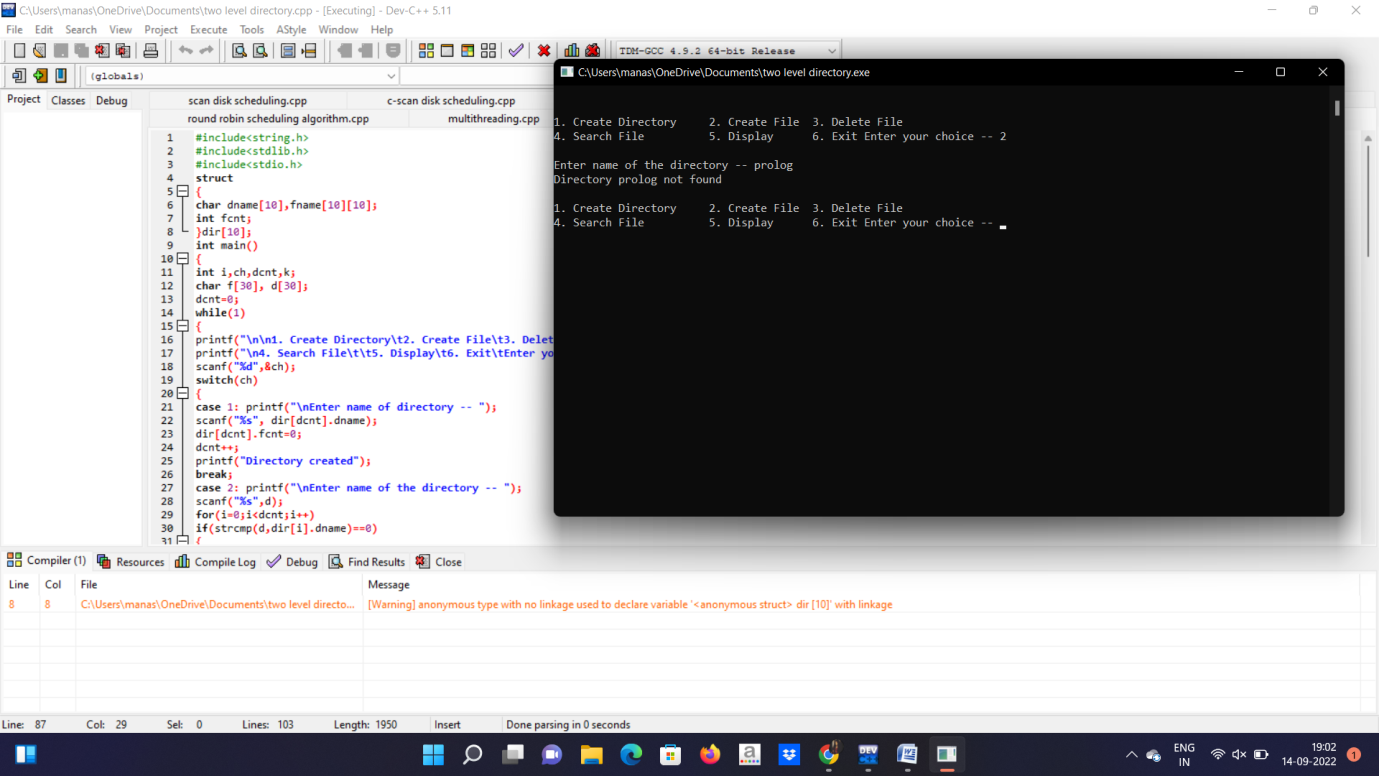
**printf("Id: %d" ,emp.empId);**

**printf("Salary: %f\n",emp.salary);**

**return 0;**

**}**

**OUTPUT:**

****

**#program 25**

Design a C program to simulate SCAN disk scheduling algorithm.

**PROGRAM:**

**#include<stdio.h>**

**int absoluteValue(int);**

**int main()**

**{**

**int queue[25],n,headposition,i,j,k,seek=0, maxrange,**

**difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;**

**float averageSeekTime;**

**printf("Enter the maximum range of Disk: ");**

**scanf("%d",&maxrange);**

**printf("Enter the number of queue requests: ");**

**scanf("%d",&n);**

**printf("Enter the initial head position: ");**

**scanf("%d",&headposition);**

**printf("Enter the disk positions to be read(queue): ");**

**for(i=1;i<=n;i++)**

**{**

**scanf("%d",&temp);**

**if(temp>headposition)**

**{**

**queue1[temp1]=temp;**

**temp1++;**

**}**

**else**

**{**

**queue2[temp2]=temp;**

**temp2++;**

**}**

**}**

**for(i=0;i<temp1-1;i++)**

**{**

**for(j=i+1;j<temp1;j++)**

**{**

**if(queue1[i]>queue1[j])**

**{**

**temp=queue1[i];**

**queue1[i]=queue1[j];**

**queue1[j]=temp;**

**}**

**}**

**}**

**for(i=0;i<temp2-1;i++)**

**{**

**for(j=i+1;j<temp2;j++)**

**{**

**if(queue2[i]<queue2[j])**

**{**

**temp=queue2[i];**

**queue2[i]=queue2[j];**

**queue2[j]=temp;**

**}**

**}**

**}**

**for(i=1,j=0;j<temp1;i++,j++)**

**{**

**queue[i]=queue1[j];**

**}**

**queue[i]=maxrange;**

**{**

**queue[i]=queue2[j];**

**}**

**queue[i]=0;**

**queue[0]=headposition;**

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

**{**

**difference = absoluteValue(queue[j+1]-queue[j]);**

**seek = seek + difference;**

**printf("Disk head moves from position %d to %d with Seek %d \n", queue[j], queue[j+1], difference);**

**}**

**averageSeekTime = seek/(float)n;**

**printf("Total Seek Time= %d\n", seek);**

**printf("Average Seek Time= %f\n", averageSeekTime);**

**}**

**int absoluteValue(int x)**

**{**

**if(x>0)**

**{**

**return x;**

**}**

**else**

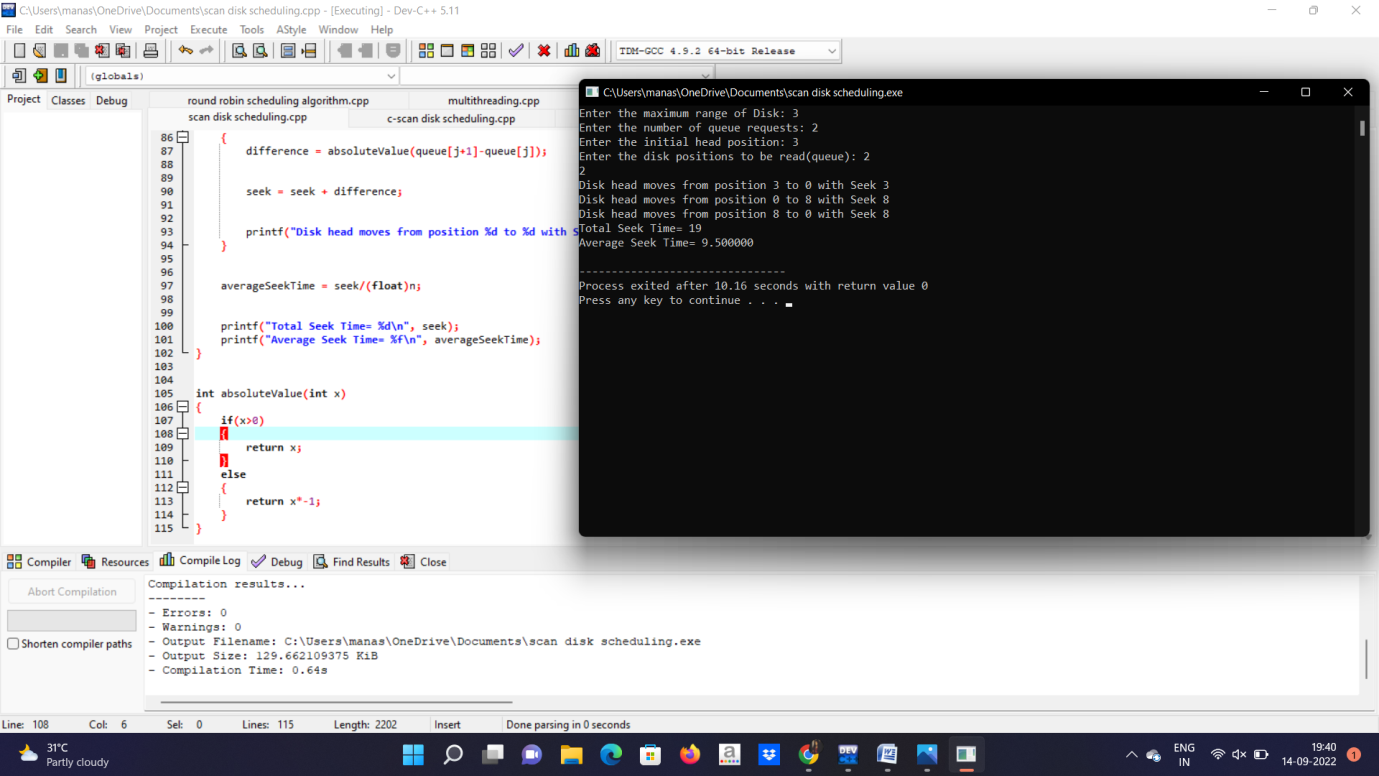
**{**

**return x\*-1;**

**}**

**}**

**OUTPUT:**

****

**#program 26**

Develop a C program to simulate C-SCAN disk scheduling algorithm.

**PROGRAM:**

#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];

}

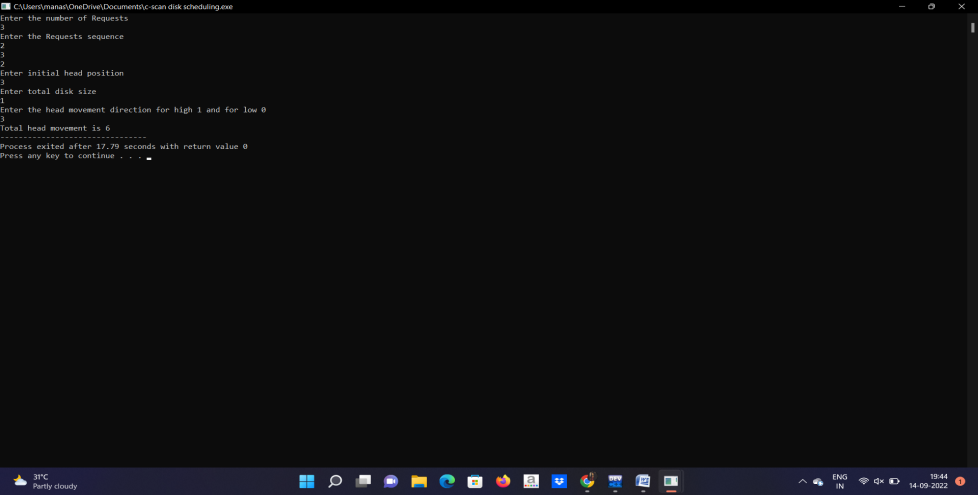
}

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

return 0;

}

**OUTPUT:**

**l**