**Dining philosopher problem:**

**Source Code :**

#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);

}

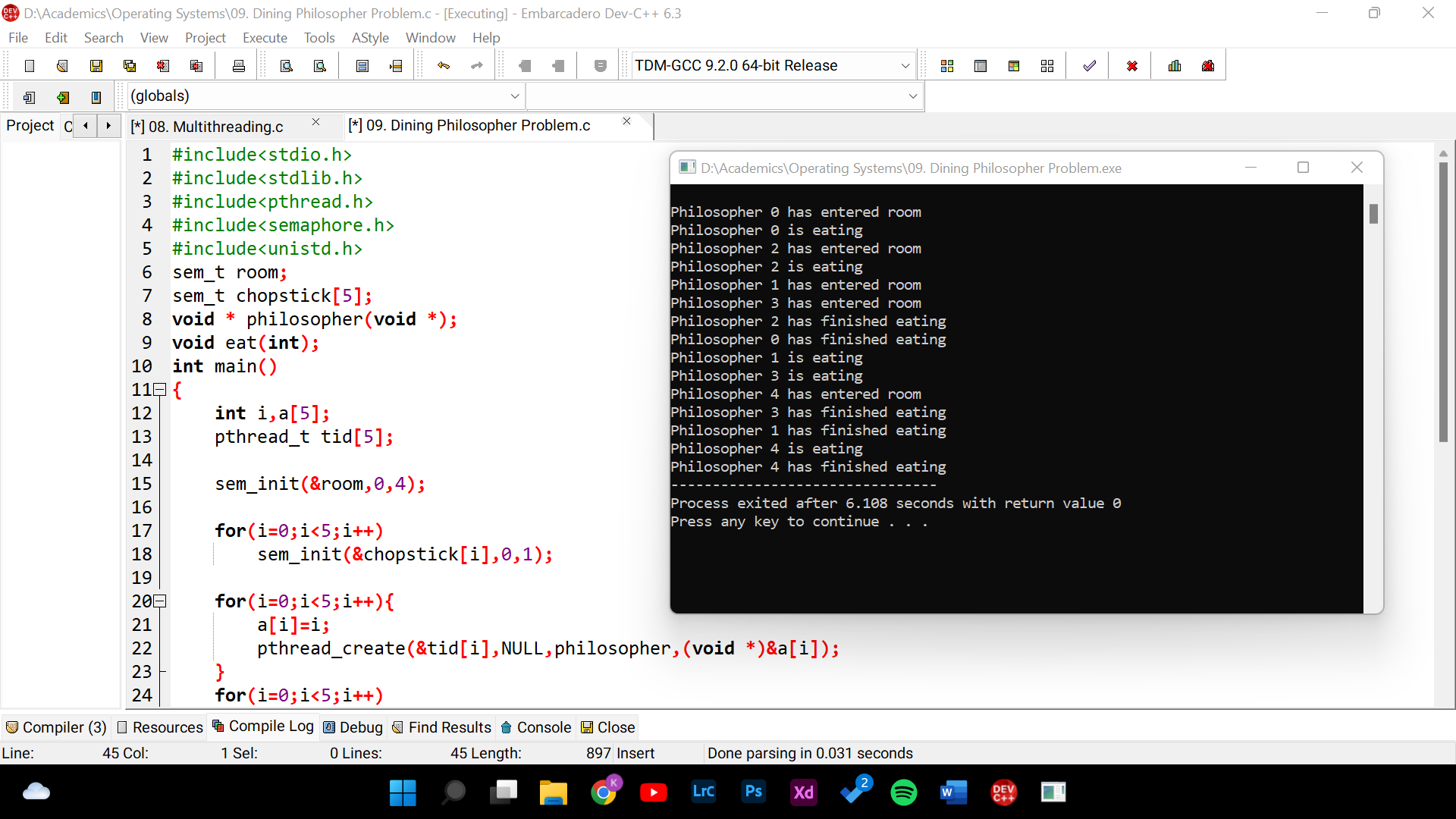
void eat(int phil)

{

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

}

**Output :**



**FIFO Paging Technique:**

**Source Code :**

#include <stdio.h>

int main()

{

int incomingStream[] = {4, 1, 2, 4, 5};

int pageFaults = 0;

int frames = 3;

int m, n, s, pages;

pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

printf("Incoming \t\t Frame 1 \t\t Frame 2 \t\t Frame 3");

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(incomingStream[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

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

{

temp[m] = incomingStream[m];

}

else if(s == 0)

{

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

}

printf("\n");

printf("%d\t\t\t",incomingStream[m]);

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

{

if(temp[n] != -1)

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

else

printf(" - \t\t\t");

}

}

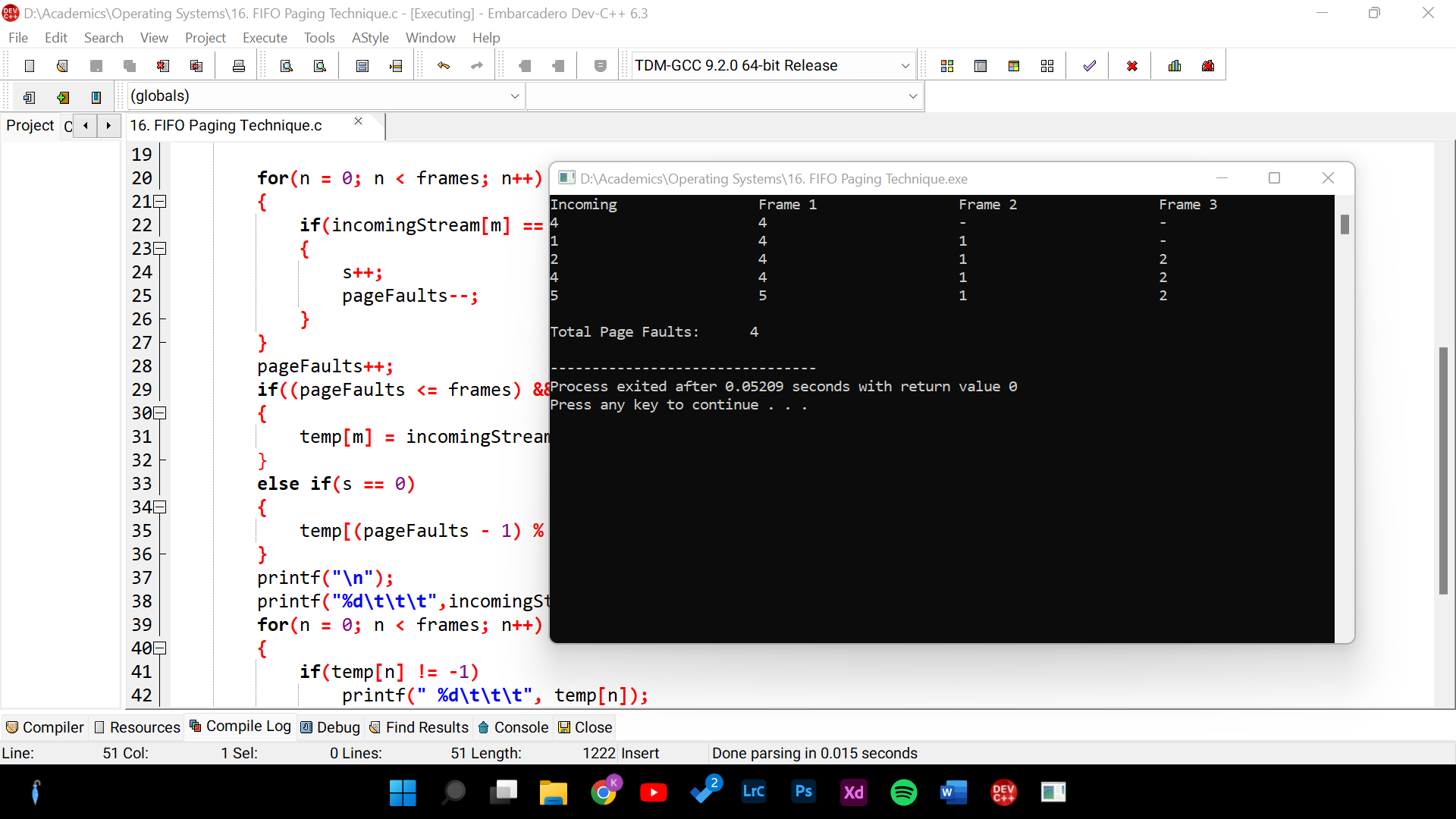
printf("\n");

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

return 0;

}

**Output :**



**First fit Strategy:**

**Source Code :**

#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]);

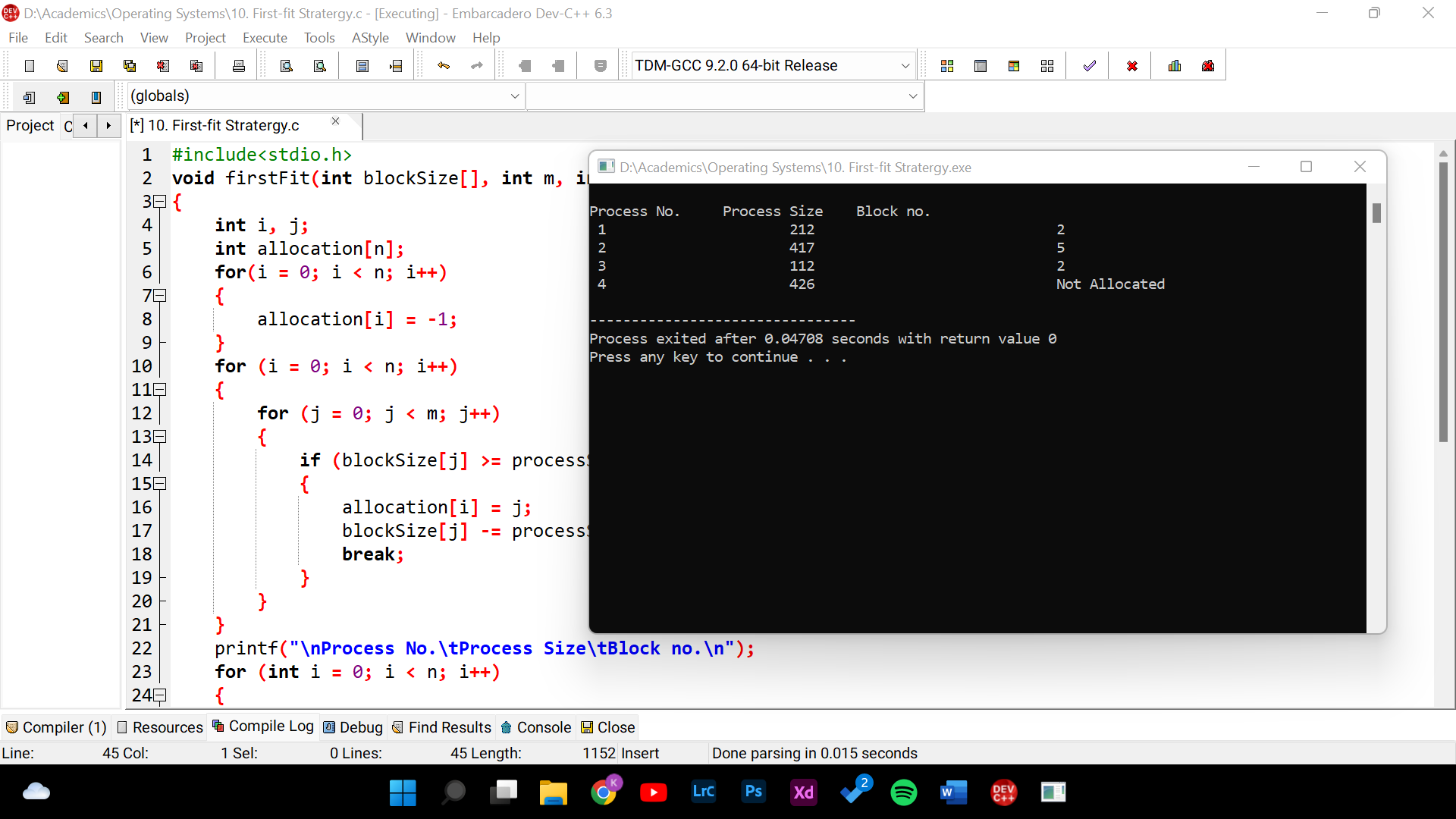
n = sizeof(processSize) / sizeof(processSize[0]);

firstFit(blockSize, m, processSize, n);

return 0 ;

}

**Output :**



**LRU Paging Technique:**

**Source Code :**

#include<stdio.h>

#include<limits.h>

int checkHit(int incomingPage, int queue[], int occupied)

{

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

{

if(incomingPage == queue[i])

return 1;

}

return 0;

}

void printFrame(int queue[], int occupied)

{

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

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

}

int main()

{

int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3};

int n = sizeof(incomingStream)/sizeof(incomingStream[0]);

int frames = 3;

int queue[n];

int distance[n];

int occupied = 0;

int pagefault = 0;

printf("Page\t\tFrame1 \t\t\tFrame2 \t\t\tFrame3\n");

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

{

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

if(checkHit(incomingStream[i], queue, occupied))

{

printFrame(queue, occupied);

}

else if(occupied < frames)

{

queue[occupied] = incomingStream[i];

pagefault++;

occupied++;

printFrame(queue, occupied);

}

else

{

int max = INT\_MIN;

int index;

for (int j = 0; j < frames; j++)

{

distance[j] = 0;

for(int k = i - 1; k >= 0; k--)

{

++distance[j];

if(queue[j] == incomingStream[k])

break;

}

if(distance[j] > max)

{

max = distance[j];

index = j;

}

}

queue[index] = incomingStream[i];

printFrame(queue, occupied);

pagefault++;

}

printf("\n");

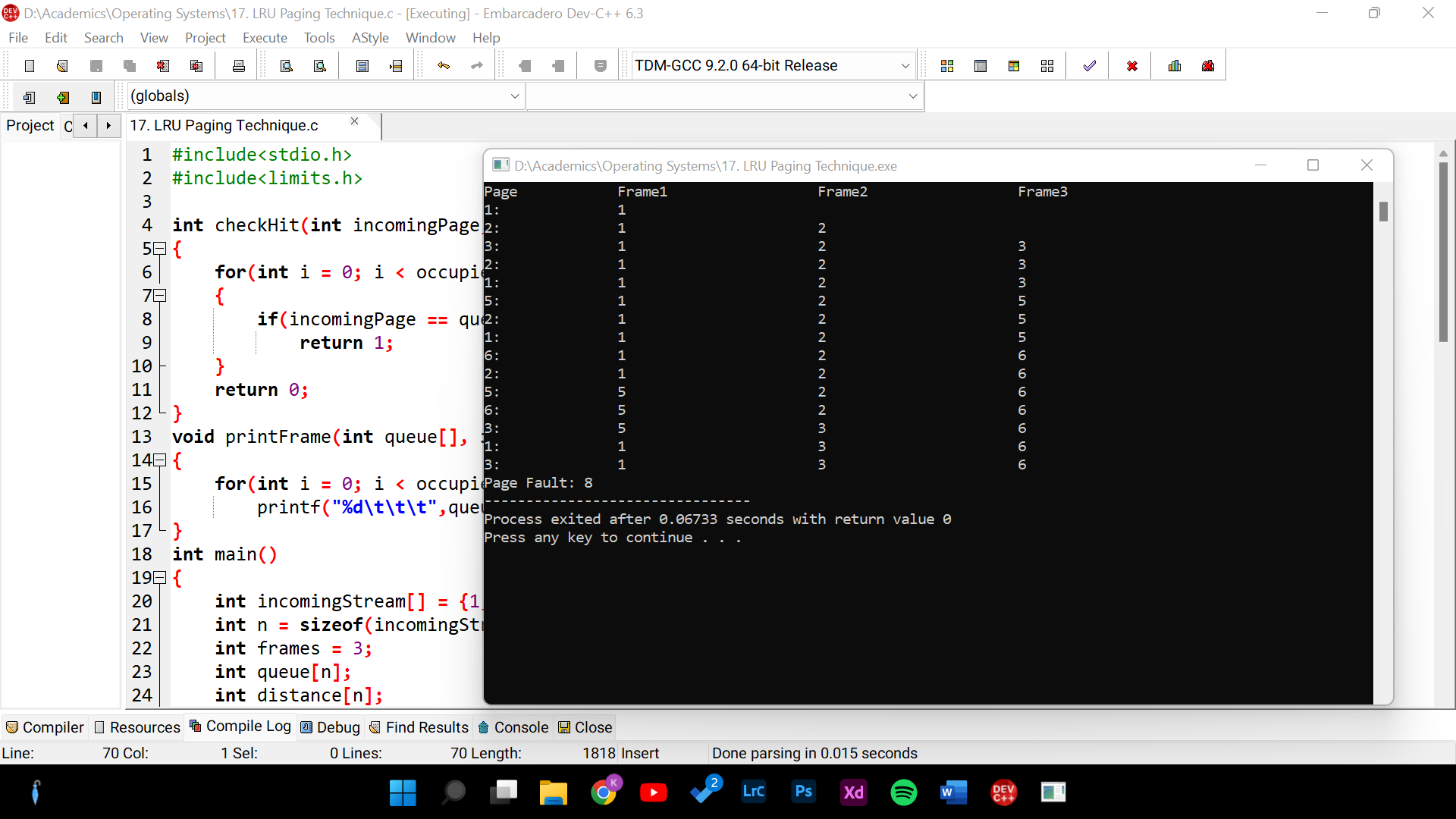
}

printf("Page Fault: %d",pagefault);

return 0;

}

**Output :**



**Multithreading:**

**Source Code :**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

void \*myThreadFun(void \*vargp)

{

sleep(1);

printf("Printing GeeksQuiz from Thread \n");

return NULL;

}

int main()

{

pthread\_t thread\_id;

printf("Before Thread\n");

pthread\_create(&thread\_id, NULL, myThreadFun, NULL);

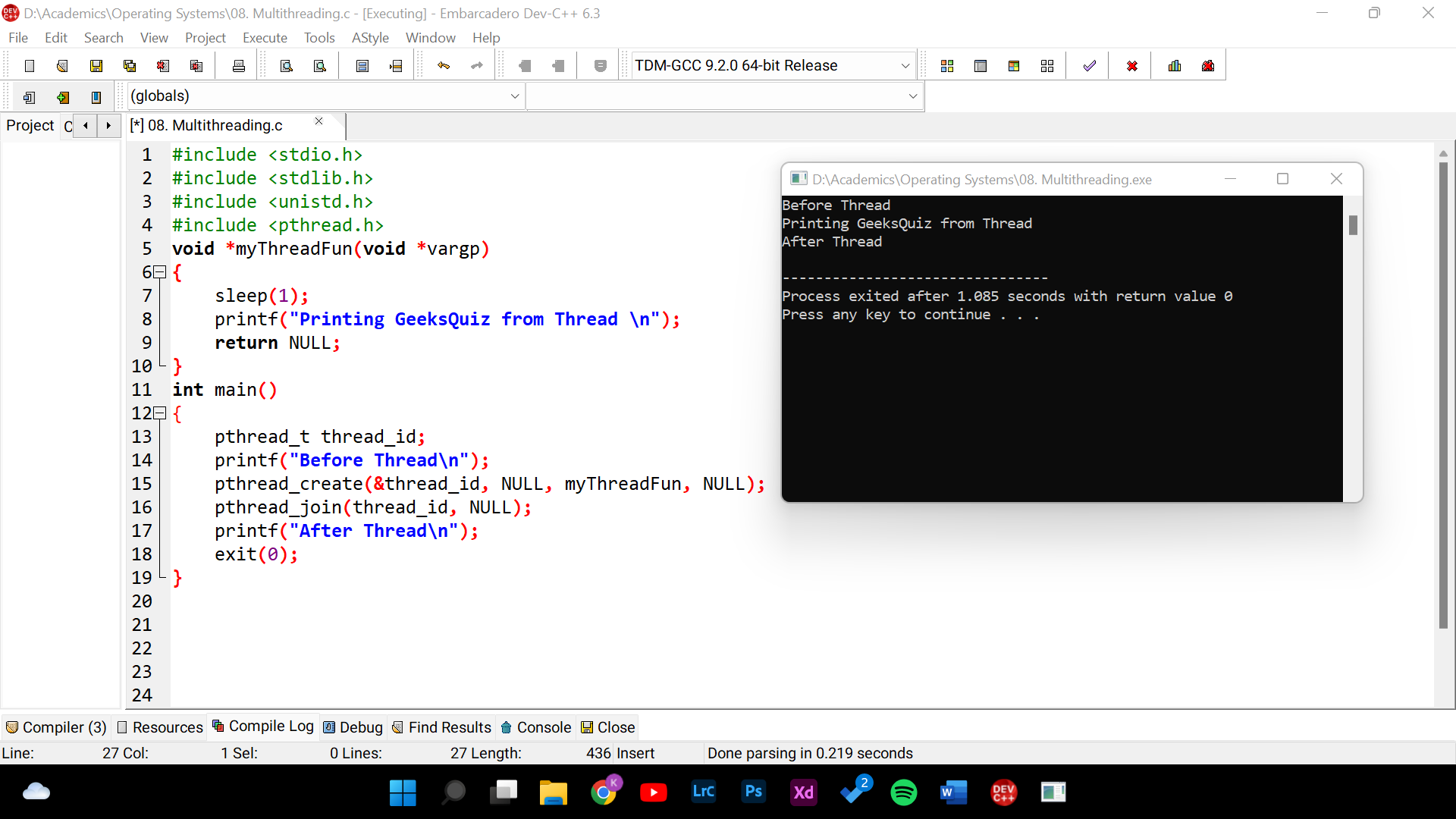
pthread\_join(thread\_id, NULL);

printf("After Thread\n");

exit(0);

}

**Output :**



**Round Robin CPU Scheduling:**

**Source Code :**

#include<stdio.h>

#include<conio.h>

void main()

{

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP;

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

{

printf("\nEnter the Arrival and Burst time of the Process[%d]\n", i+1);

printf("Arrival time is: ");

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

printf("\nBurst time is: ");

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

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

{

y--;

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else {

i=0;

}

}

avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

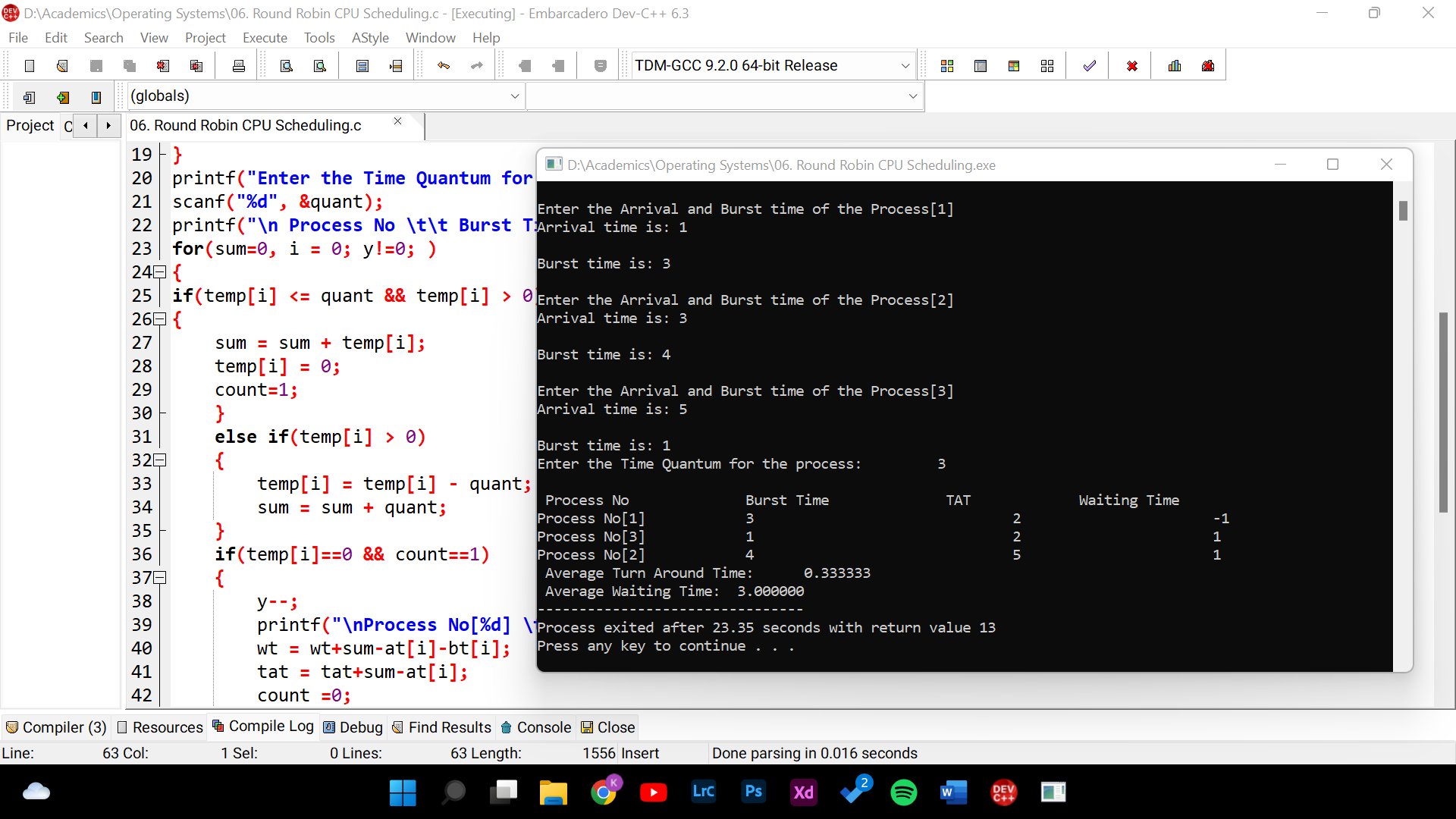
printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();

}

**Output :**



**Single level Directory:**

**Source Code :**

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

int nf=0,i=0,j=0,ch;

char mdname[10],fname[10][10],name[10];

printf("Enter the directory name:");

scanf("%s",mdname);

printf("Enter the number of files:");

scanf("%d",&nf);

do

{

printf("Enter file name to be created:");

scanf("%s",name);

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

{

if(!strcmp(name,fname[i]))

break;

}

if(i==nf)

{

strcpy(fname[j++],name);

nf++;

}

else

printf("There is already %s\n",name);

printf("Do you want to enter another file(yes - 1 or no - 0):");

scanf("%d",&ch);

}

while(ch==1);

printf("Directory name is:%s\n",mdname);

printf("Files names are:");

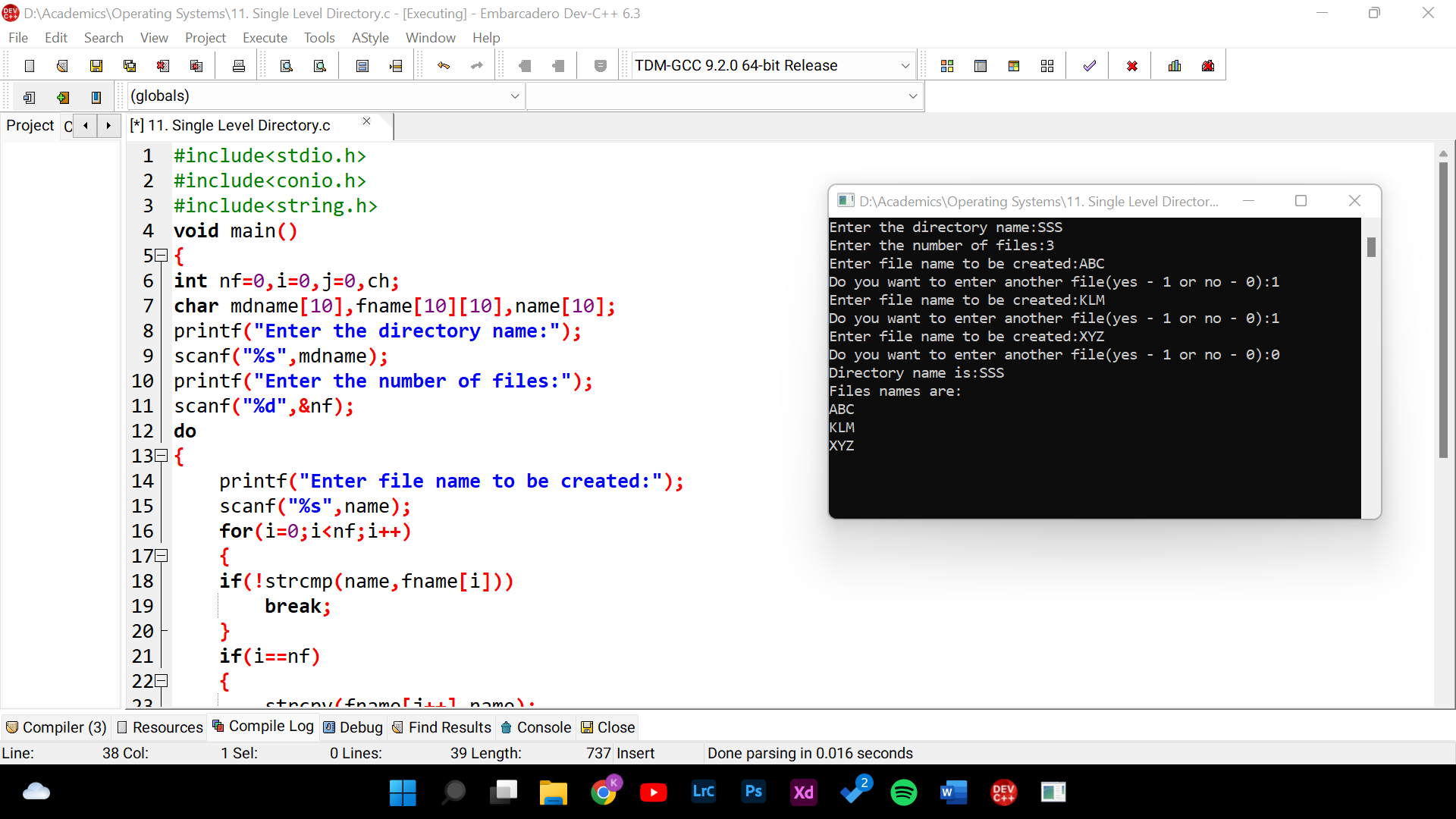
for(i=0;i<j;i++)

printf("\n%s",fname[i]);

getch();

}

**Output:**



**Two level Directory:**

**Source Code :**

#include<string.h>

#include<stdlib.h>

#include<stdio.h>

struct

{

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

int fcnt;

}dir[10];

void 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");

printf("\nEnter 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;

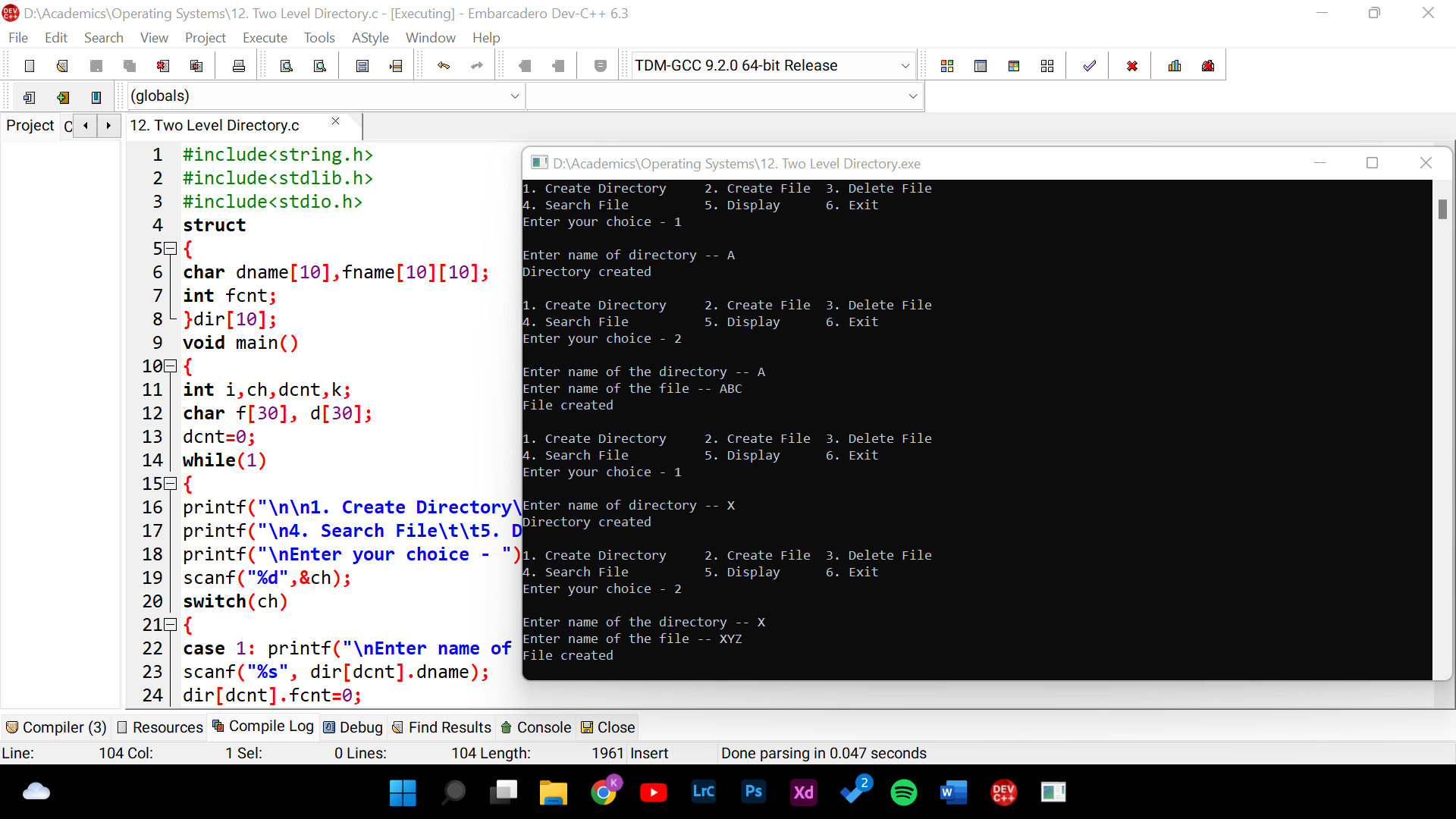
default:exit(0);

}

}

}

**Output :**



**Highest priority to execute next:**

**Source Code :**

#include<stdio.h>

struct priority\_scheduling {

char process\_name;

int burst\_time;

int waiting\_time;

int turn\_around\_time;

int priority;

};

int main() {

int number\_of\_process;

int total = 0;

struct priority\_scheduling temp\_process;

int ASCII\_number = 65;

int position;

float average\_waiting\_time;

float average\_turnaround\_time;

printf("Enter the total number of Processes: ");

scanf("%d", & number\_of\_process);

struct priority\_scheduling process[number\_of\_process];

printf("\nPlease Enter the Burst Time and Priority of each process:\n");

for (int i = 0; i < number\_of\_process; i++) {

process[i].process\_name = (char) ASCII\_number;

printf("\nEnter the details of the process %c \n", process[i].process\_name);

printf("Enter the burst time: ");

scanf("%d", & process[i].burst\_time);

printf("Enter the priority: ");

scanf("%d", & process[i].priority);

ASCII\_number++;

}

for (int i = 0; i < number\_of\_process; i++) {

position = 'I';

for (int j = i + 1; j < number\_of\_process; j++) {

if (process[j].priority > process[position].priority)

position = j;

}

temp\_process = process[i];

process[i] = process[position];

process[position] = temp\_process;}

process[0].waiting\_time = 0;

for (int i = 1; i < number\_of\_process; i++) {

process[i].waiting\_time = 0;

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

process[i].waiting\_time += process[j].burst\_time;

}

total += process[i].waiting\_time;

average\_waiting\_time = (float) total / (float) number\_of\_process;

total = 0;

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

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

for (int i = 0; i < number\_of\_process; i++) {

process[i].turn\_around\_time = process[i].burst\_time + process[i].waiting\_time;

total += process[i].turn\_around\_time;

printf("\t %c \t\t %d \t\t %d \t\t %d", process[i].process\_name, process[i].burst\_time,

process[i].waiting\_time, process[i].turn\_around\_time);

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

}

average\_turnaround\_time = (float) total / (float) number\_of\_process;

printf("\n\n Average Waiting Time : %f", average\_waiting\_time);

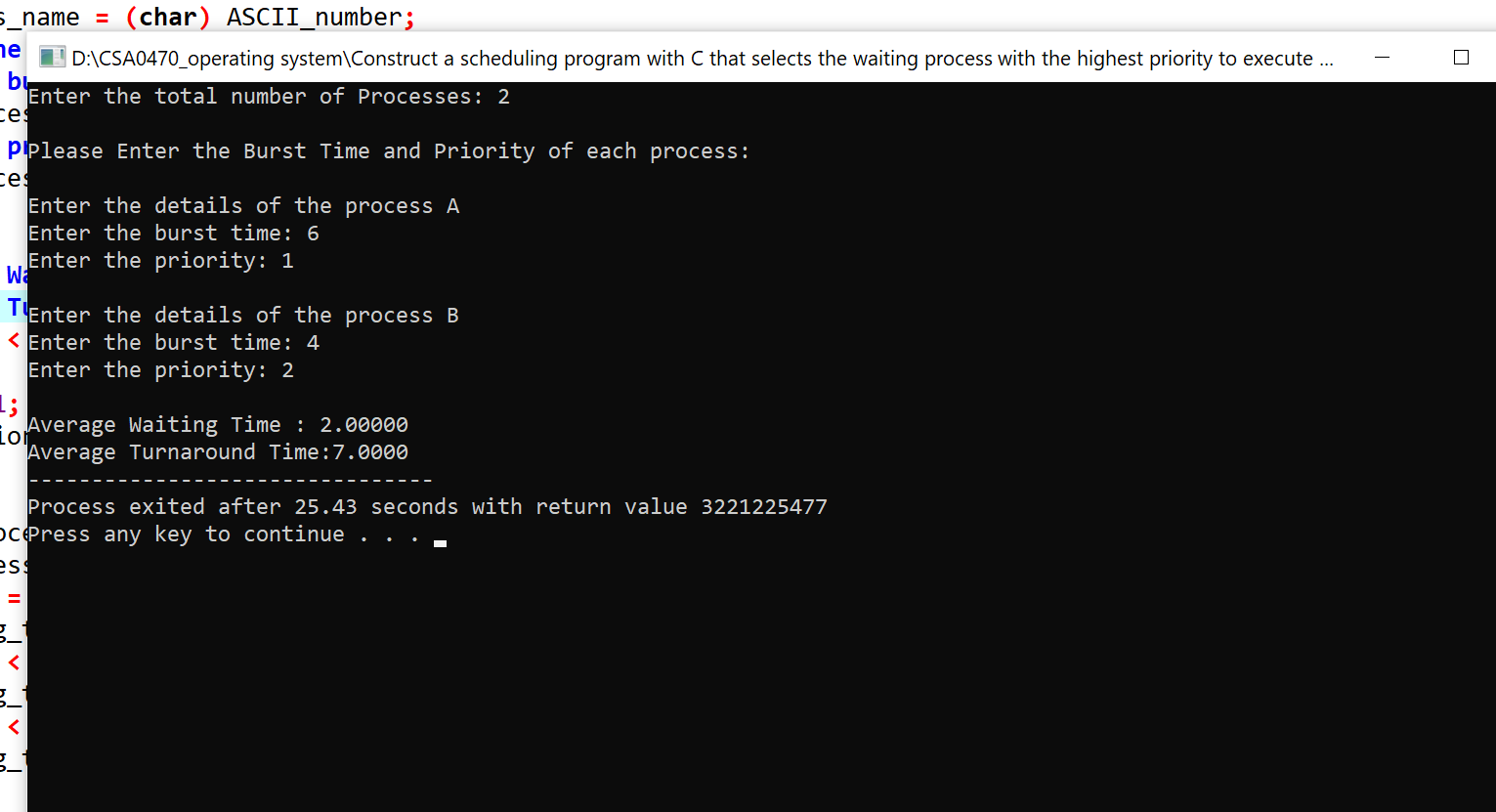
printf("\n Average Turnaround Time: %f\n", average\_turnaround\_time);

return 0;

}

}

**Output :**



**Identify the system calls:**

**Source Code :**

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE \*fptr1, \*fptr2;

char filename[100], c;

printf("Enter the filename to open for reading \n");

scanf("%s", filename);

// Open one file for reading

fptr1 = fopen(filename, "r");

if (fptr1 == NULL)

{

printf("Cannot open file %s \n", filename);

exit(0);

}

printf("Enter the filename to open for writing \n");

scanf("%s", filename);

// Open another file for writing

fptr2 = fopen(filename, "w");

if (fptr2 == NULL)

{

printf("Cannot open file %s \n", filename);

exit(0);

}

// Read contents from file

c = fgetc(fptr1);

while (c != EOF)

{

fputc(c, fptr2);

c = fgetc(fptr1);

}

printf("\nContents copied to %s", filename);

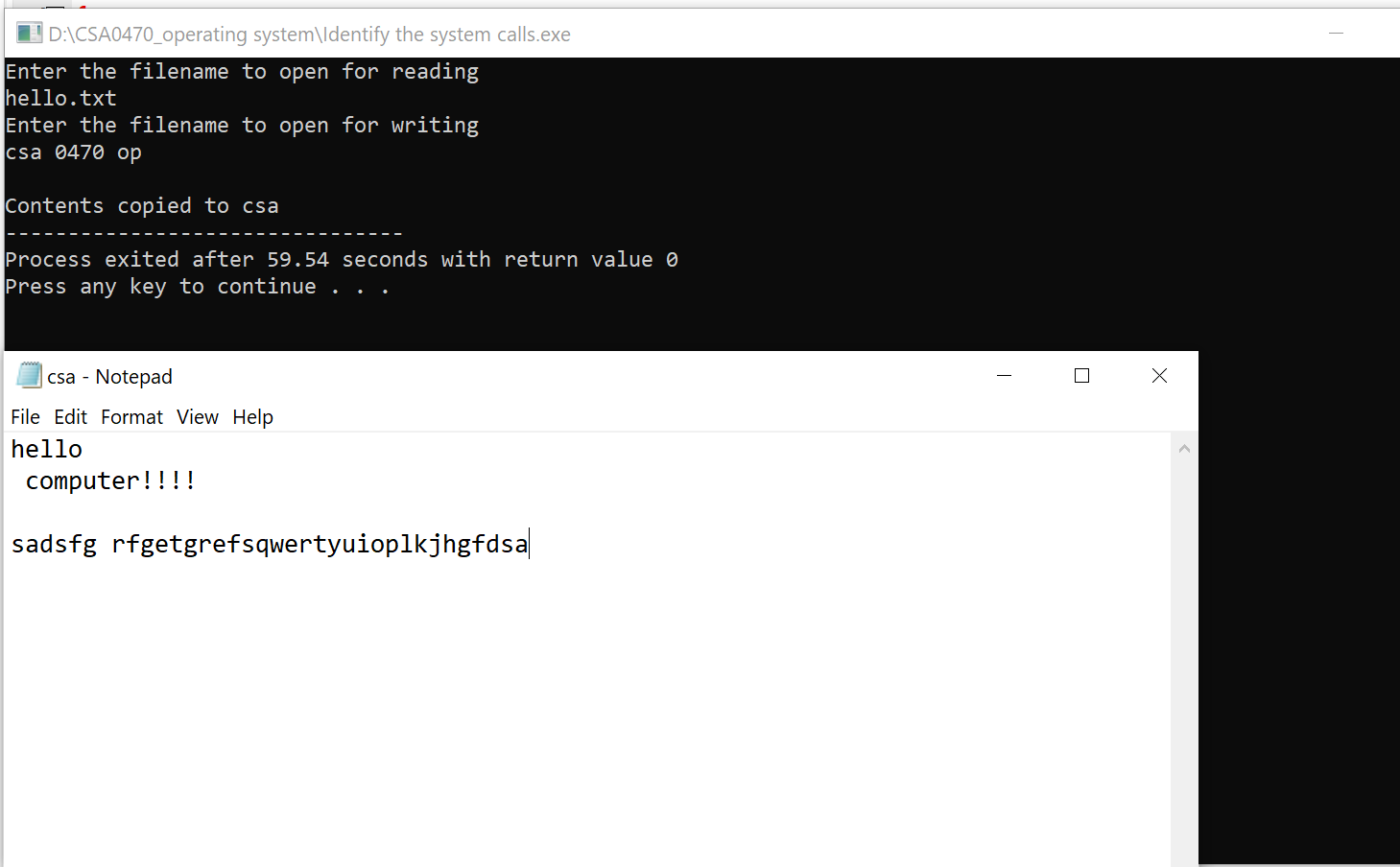
fclose(fptr1);

fclose(fptr2);

return 0;

}

**Output :**



**Smallest execution time to execute next:**

**Source Code :**

#include<stdio.h>

int 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:");

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

{

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

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

p[i]=i+1;

}

//sorting of burst times

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("\nProcesst Burst Time ,Waiting Time ,Turnaround Time:");

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

{

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

total+=tat[i];

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

}

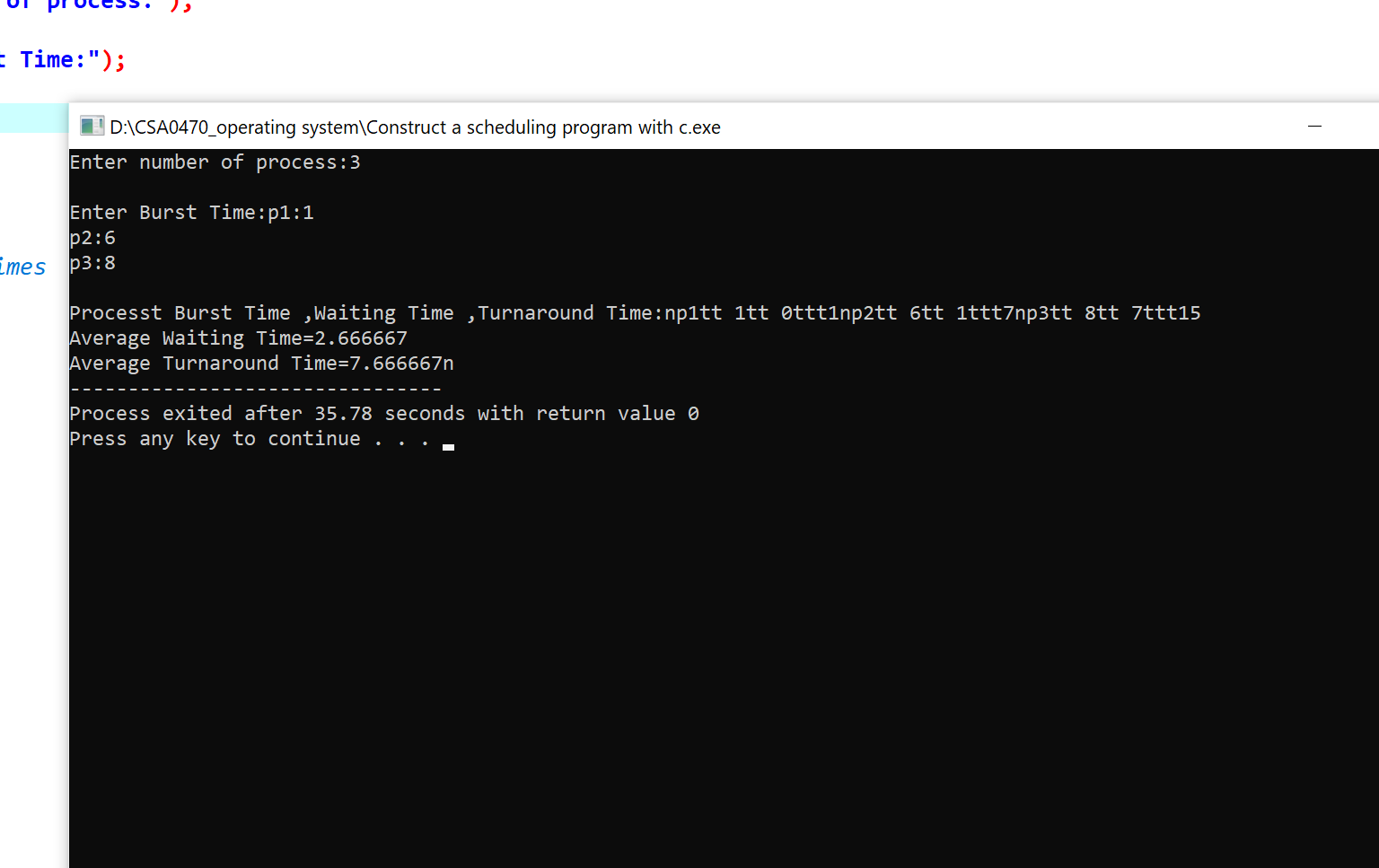
avg\_tat=(float)total/n;

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

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

}

**Output :**



**First Come First Served technique:**

**Source Code :**

#include<stdio.h>

void findWaitingTime(int processes[], int n,int bt[], int wt[])

{

wt[0] = 0;

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

wt[i] = bt[i-1] + wt[i-1] ;

}

void findTurnAroundTime( int processes[], int n,

int bt[], int wt[], int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

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

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

}

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst time Waiting time Turn around time\n");

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

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

printf(" %d ",(i+1));

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

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

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

}

int s=(float)total\_wt / (float)n;

int t=(float)total\_tat / (float)n;

printf("Average waiting time = %d",s);

printf("\n");

printf("Average turn around time = %d ",t);

}

int main()

{

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

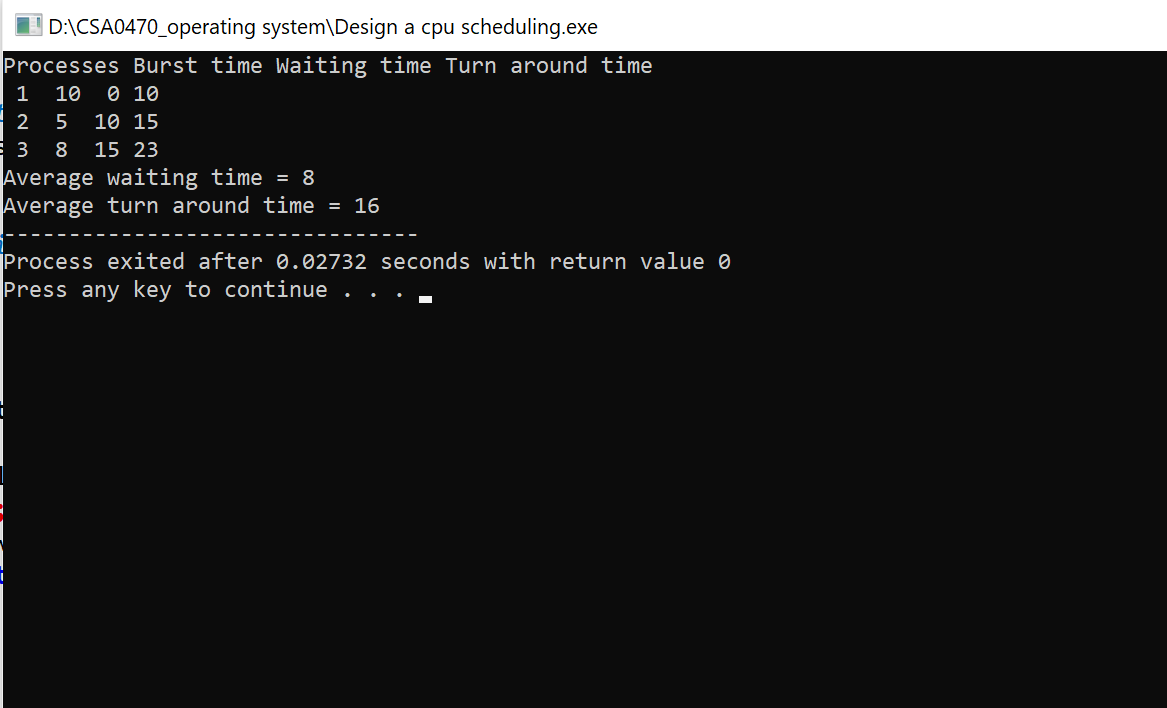
int burst\_time[] = {10, 5, 8};

findavgTime(processes, n, burst\_time);

return 0;

}

**Output :**



**Simulate the optimal paging technique of memory management:**

**Source Code :**

#include<stdio.h>

#include<conio.h>

void main()

{

int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;

int s[10], fno[10][20];

printf("\nEnter the memory size -- ");

scanf("%d",&ms);

printf("\nEnter the page size -- ");

scanf("%d",&ps);

nop = ms/ps;

printf("\nThe no. of pages available in memory are -- %d ",nop);

printf("\nEnter number of processes -- ");

scanf("%d",&np);

rempages = nop;

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

{

printf("\nEnter no. of pages required for p[%d]-- ",i);

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

if(s[i] >rempages)

{

printf("\nMemory is Full");

break;

}

rempages = rempages - s[i];

printf("\nEnter pagetable for p[%d] --- ",i);

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

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

}

printf("\nEnter Logical Address to find Physical Address ");

printf("\nEnter process no. and pagenumber and offset -- ");

scanf("%d %d %d",&x,&y, &offset);

if(x>np || y>=s[i] || offset>=ps)

printf("\nInvalid Process or Page Number or offset");

else

{ pa=fno[x][y]\*ps+offset;

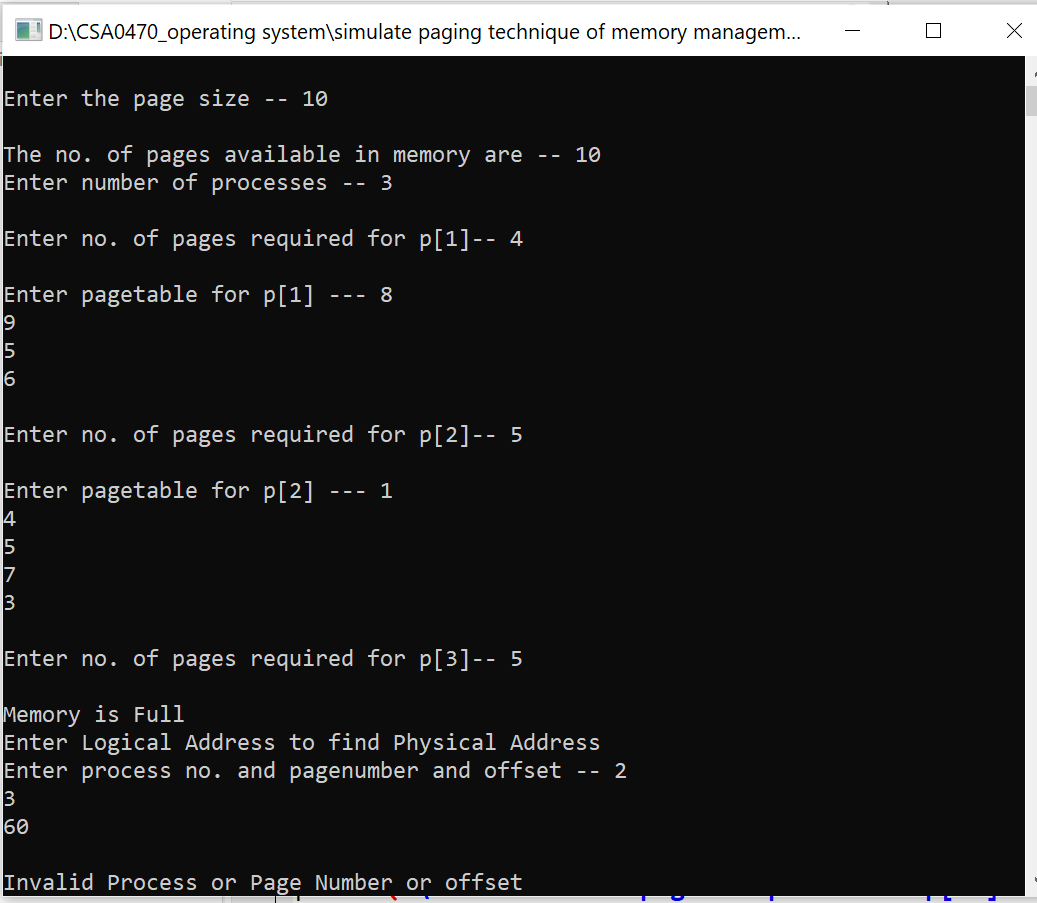
printf("\nThe Physical Address is -- %d",pa);

}

getch();

}

**Output :**



Rondom Access File:

**Source Code :**

#include<stdio.h>

int main()

{

FILE \*fp;

fp=fopen("hello.txt","r");

if(!fp)

{

printf("Error: File cannot be opened\n") ;

return 0;

}

//Since the file pointer points to the starting of the file, ftell() will return 0

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

char ch;

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

{

//Here, we traverse the entire file and print its contents until we reach its end.

printf("%c",ch);

}

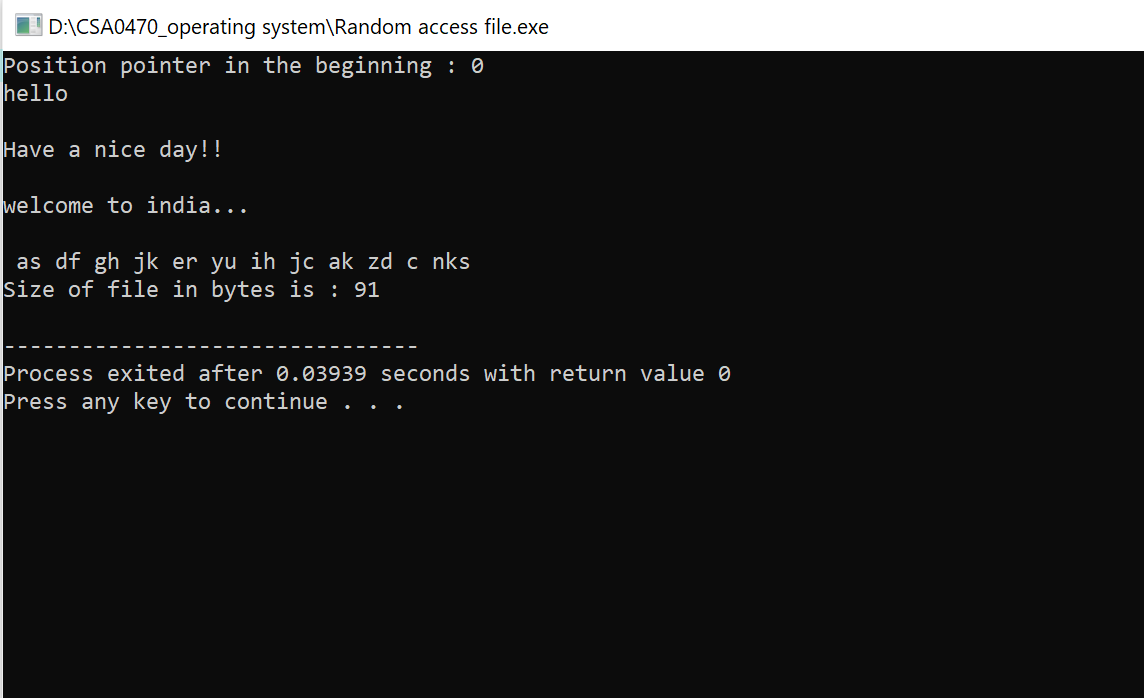
printf("\nSize of file in bytes is : %ld\n",ftell(fp));

fclose(fp);

return 0;

}

**Output :**



**Bankers algorithm:**

**Source Code :**

#include <stdio.h>

int main()

{

int n, m, i, j, k;

n = 5;

m = 3;

int alloc[5][3] = { { 0, 1, 0 },

{ 2, 0, 0 },

{ 3, 0, 2 },

{ 2, 1, 1 },

{ 0, 0, 2 } };

int max[5][3] = { { 7, 5, 3 },

{ 3, 2, 2 },

{ 9, 0, 2 },

{ 2, 2, 2 },

{ 4, 3, 3 } };

int avail[3] = { 3, 3, 2 };

int f[n], ans[n], ind = 0;

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

f[k] = 0;

}

int need[n][m];

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

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

need[i][j] = max[i][j] - alloc[i][j];

}

int y = 0;

for (k = 0; k < 5; k++) {

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

if (f[i] == 0) {

int flag = 0;

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

if (need[i][j] > avail[j]){

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

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

avail[y] += alloc[i][y];

f[i] = 1;

}

}

}

}

int flag = 1;

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

{

if(f[i]==0)

{

flag=0;

printf("The following system is not safe");

break;

}

}

if(flag==1)

{

printf("Following is the SAFE Sequence\n");

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

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

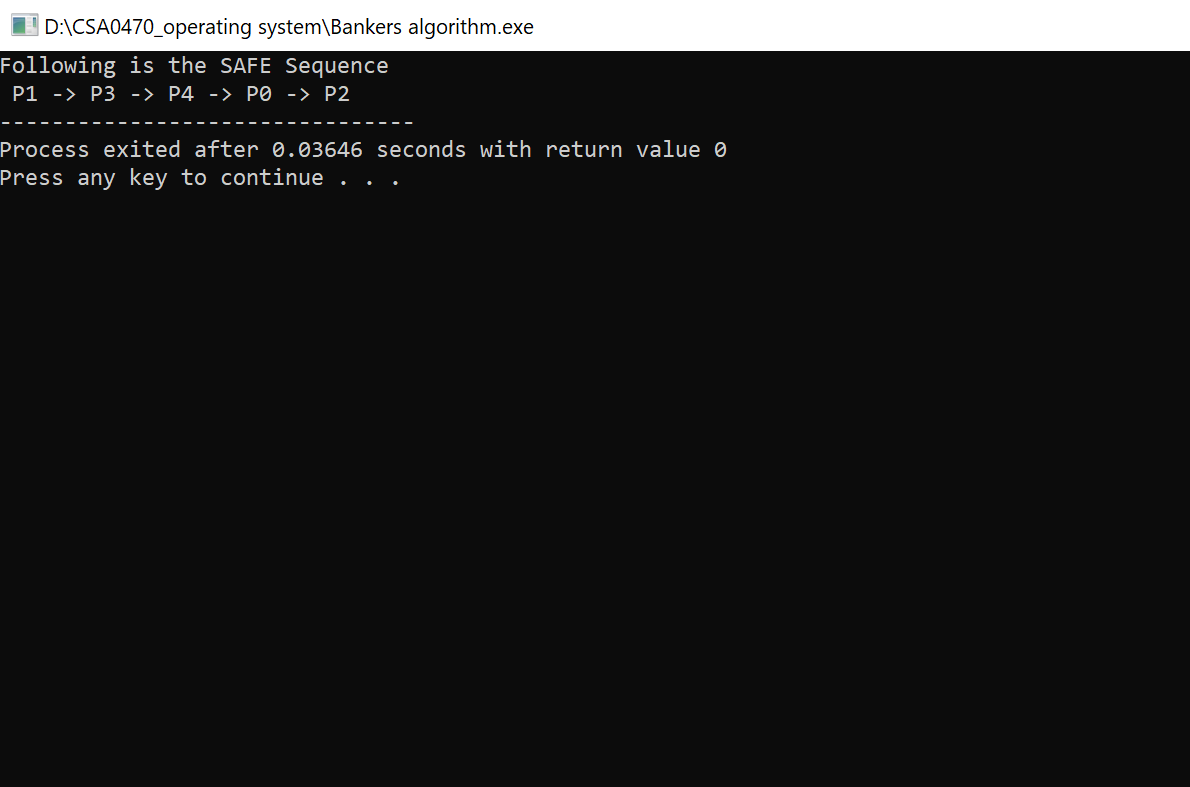
printf(" P%d", ans[n - 1]);

}

return (0);

}

**Output :**



**Producer consumer problem:**

**Source Code :**

#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;

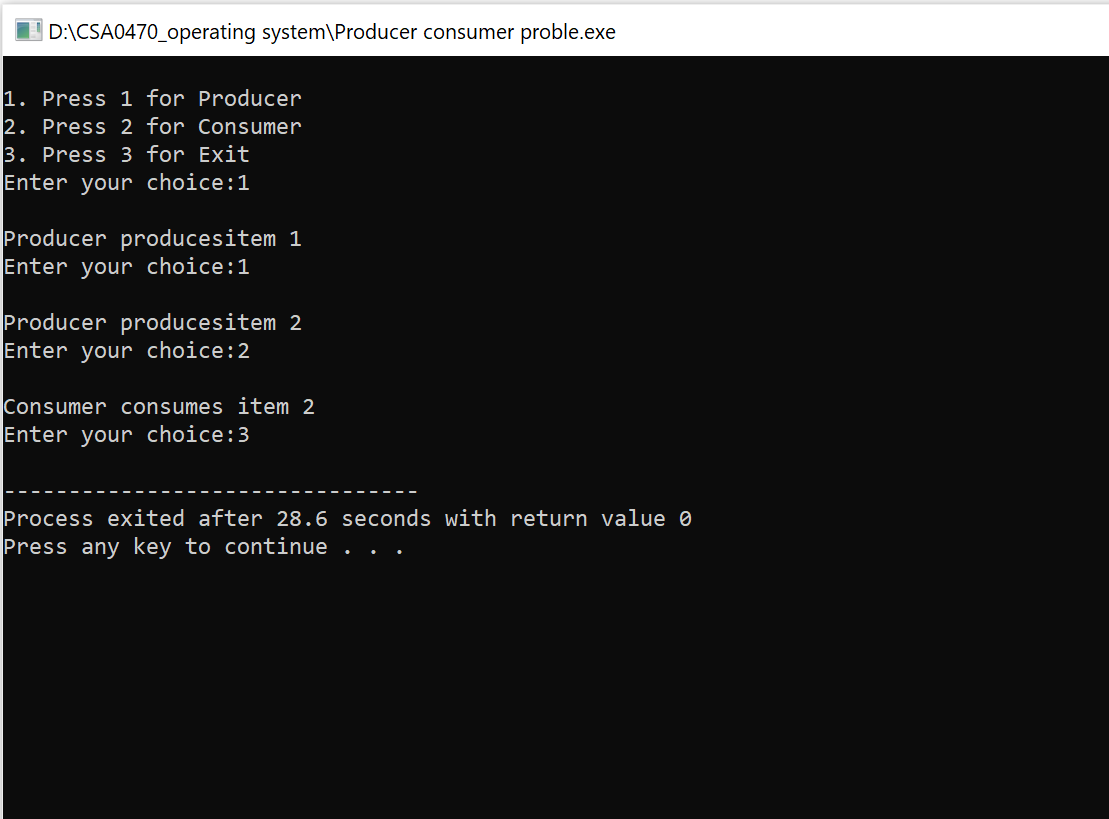
case 3:

exit(0);

break;

} } }

**Output :**



**Create a new process by invoking:**

**Source Code :**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

fork();

printf("Hello world!\n");

return 0;

}

**Output :**

