**INDEX**

| SNO | | DATE | CONTEXT | PAGE NO | | |
| --- | --- | --- | --- | --- | --- | --- |
| 1. | | 3/6/2022 | 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. | 4-7 | | |
| 2. | | 3/6/2022 | Identify the system calls to copy the content of one file to another and illustrate the same using a C program. | 8-10 | | |
| 3. | | 3/6/2022 | Design a CPU scheduling program with C using First Come First Served technique with the following considerations. a. All processes are activated at time 0. b. Assume that no process waits on I/O devices. | 11-14 | | |
| 4. | 4/6/2022 | | Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next. | | 15-18 |
| 5. | 4/6/2022 | | Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C. | | 19-24 |
| 6. | 4/6/2022 | | Construct a C program to simulate producer-consumer problem using semaphores. | | 25-30 |
| 7. | 6/6/2022 | | Construct a program to simulate the First in First Out paging technique of memory management. When a page must be replaced, the oldest page is chosen. | | 31-34 |
| 8. | 6/6/2022 | | Construct a program to simulate the Least Recently Used paging technique of memory management. When a  page must be replaced, the oldest page is chosen | | 35-41 |
| 9. | 6/6/2022 | | Construct a program to simulate the optimal paging technique of memory management. The operating system replaces the page that will not be used for the longest period of time in future. | | 42-50 |
| 10. | 7/6/2022 | | 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 | |  |
| 11. | 7/6/2022 | | 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. | |  |
| 12. | 7/6/2022 | | 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. | |  |
| 13. | 8/6/2022 | | Construct a C program to simulate the First Come First Served disk scheduling algorithm. | |  |
| 14. | 8/6/2022 | | Illustrate the various File Access Permission and different types users in Linux | |  |
| 15. | 8/6/2022 | | Study the features and deployment processes of Xen and VMware on Linux. | |  |

EXERCISE 1-

AIM:

Creating 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:

STEP 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..

STEP 2. 2.To finish a child process, the exit() system call is used in the child process.

STEP 3. The wait() function is defined in the header sys/wait.h and the exit() function is defined in the header stdlib.h.

STEP 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.

STEP 5. On the parent process wait(NULL) is used to wait for the child process to finish.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void) {

pid\_t pid = fork();

if(pid == 0) {

printf("Child => PPID: %d PID: %d\n", getppid(), getpid());

exit(EXIT\_SUCCESS);

}

else if(pid > 0) {

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

printf("Waiting for child process to finish.\n");

wait(NULL);

printf("Child process finished.\n");

}

else {

printf("Unable to create child process.\n");

}

return EXIT\_SUCCESS;

}

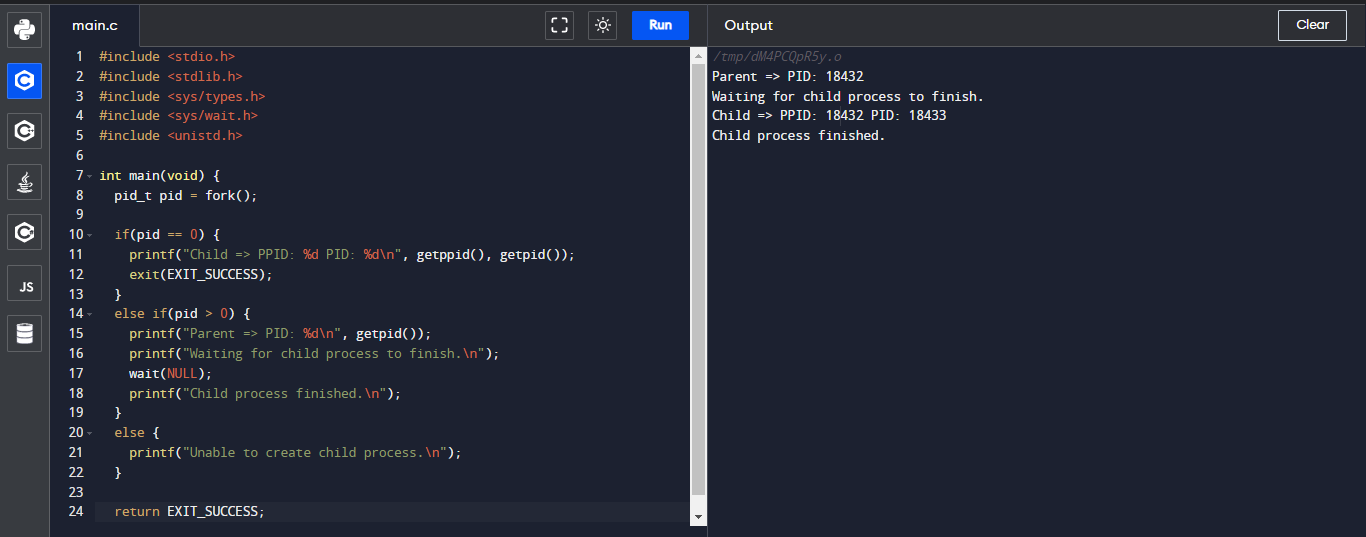
OUTPUT:

Parent => PID: 18432

Waiting for child process to finish.

Child => PPID: 18432 PID: 18433

Child process finished.



RESULT:

Thus the program has been successfully implemented using system call in c language

EXERCISE 2-

AIM:

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

ALGORITHM:

1. The header files required

2.We have used pointers in the program.

3.The pointer in C language is a variable which stores the address of another variable.

PROGRAM :

#include<stdio.h>

#include<stdlib.h>

int main()

{

FILE \*f1,\*f2;

char filename[100],c;

f1=fopen("Documents\file1.txt","r");

f2=fopen("Documents\file2.txt","w");

c=fgetc(f1);

while(c!=EOF)

{

fputc(c,f2);

c=fgetc(f1);

}

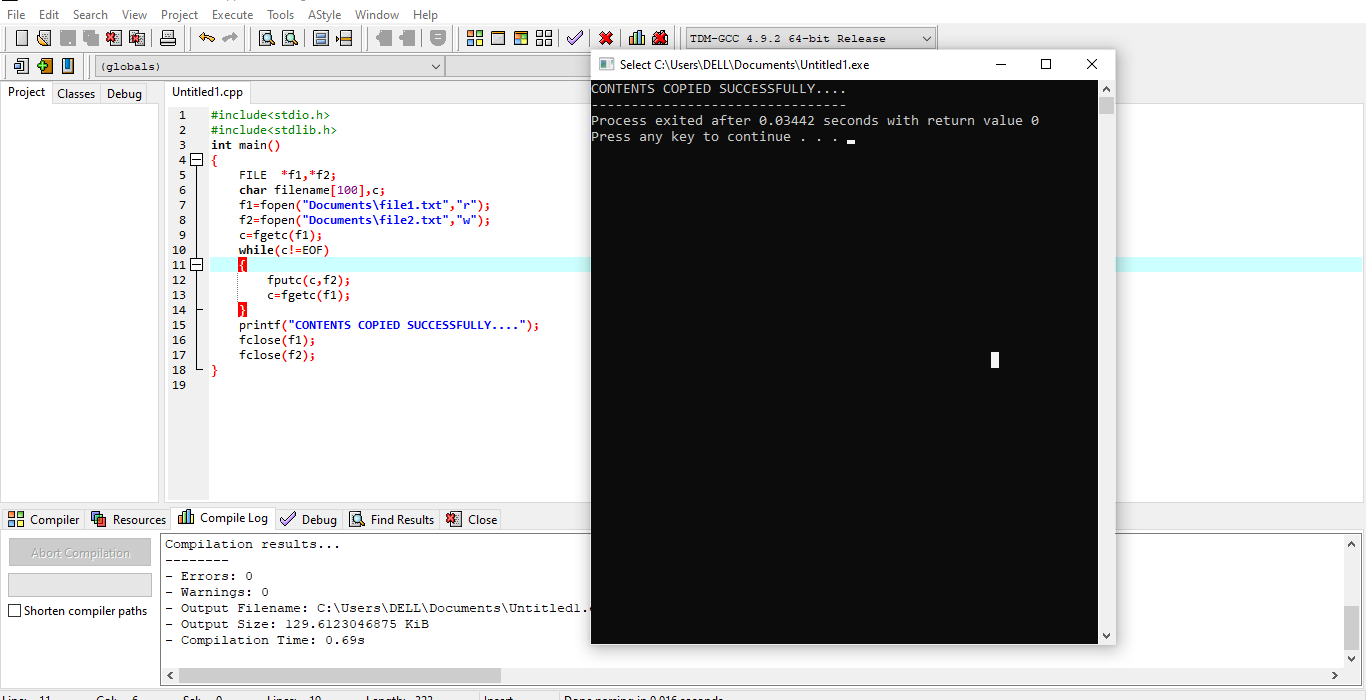
printf("CONTENTS COPIED SUCCESSFULLY....");

fclose(f1);

fclose(f2);

}

OUTPUT :



RESULT :

Thus the program of identifying the system calls to copy the contant of one to other file is successfully completed

EXERCISE 3-

AIM:

Design a CPU scheduling program with C using First Come First Served technique with the following considerations.

a. All processes are activated at time 0.

b. Assume that no process waits on I/O devices.

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

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

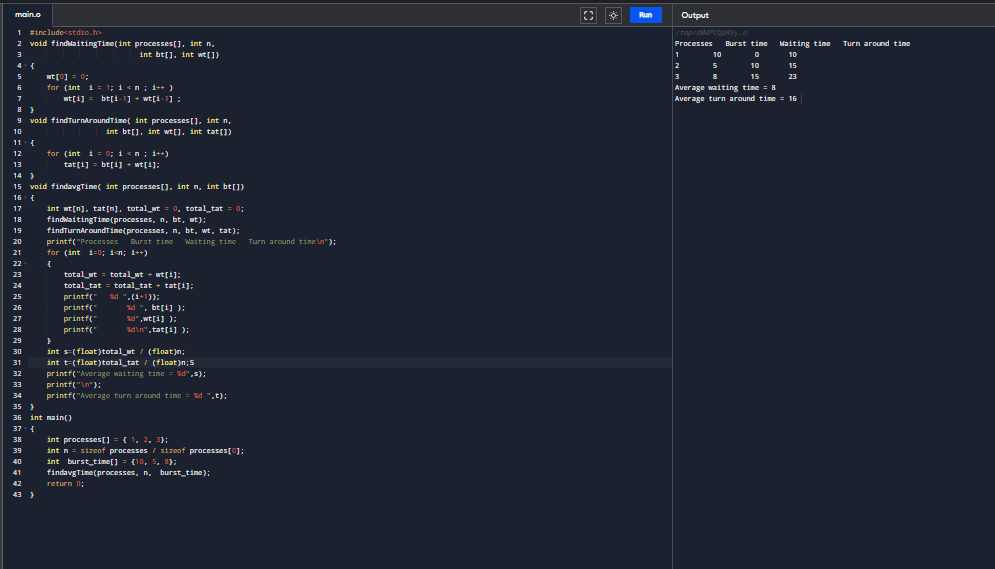
P[1] 4 0 4

P[2] 5 4 9

P[3] 6 9 15

Average Waiting Time:4

Average Turnaround Time:9



RESULTS:

Thus Designing a CPU scheduling program with C using First Come First Served technique is successfully complesd

EXERCISE 4 -

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

ALGORITHM:

In shortest job first scheduling algorithm, the processor selects the waiting process with the smallest execution time to execute next.

PROGRAM:

#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:\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

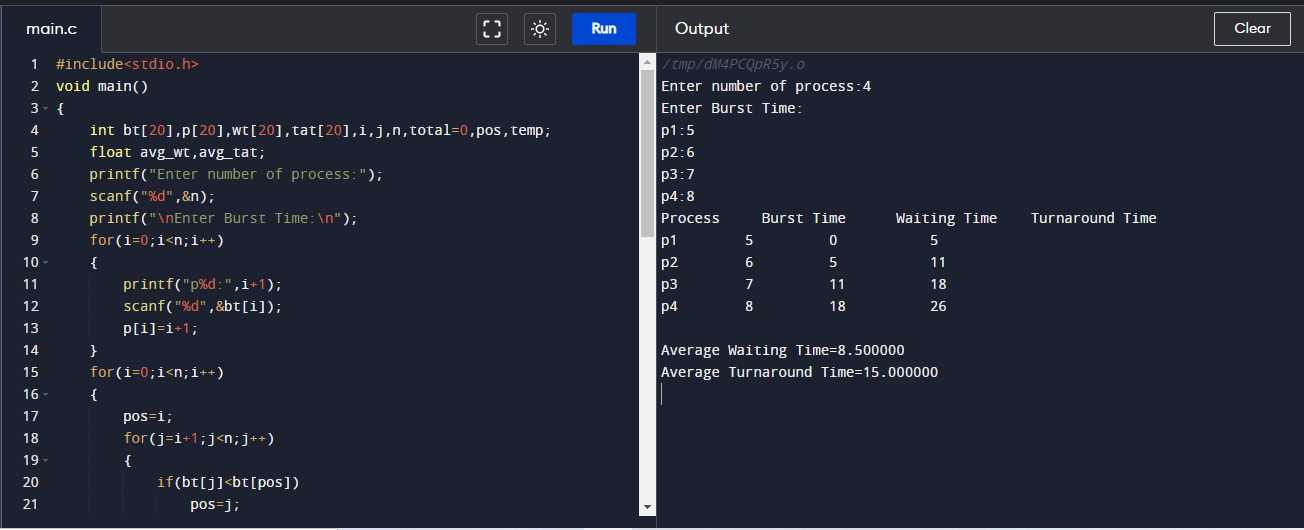
p2 6 5 11

p3 7 11 18

p4 8 18 26

Average Waiting Time=8.500000

Average Turnaround Time=15.000000



RESULT:

Thus Construction of a scheduling program with C that selects the waiting process with the smallest execution time to execute next was successfully completed.

EXERCISE 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 5 1 1 2 1 1 0 0 2

Enter the process making the new request:1 1 2

Enter the requested resource:2

Enter the process which are n blocked or running:process 3:

1 2

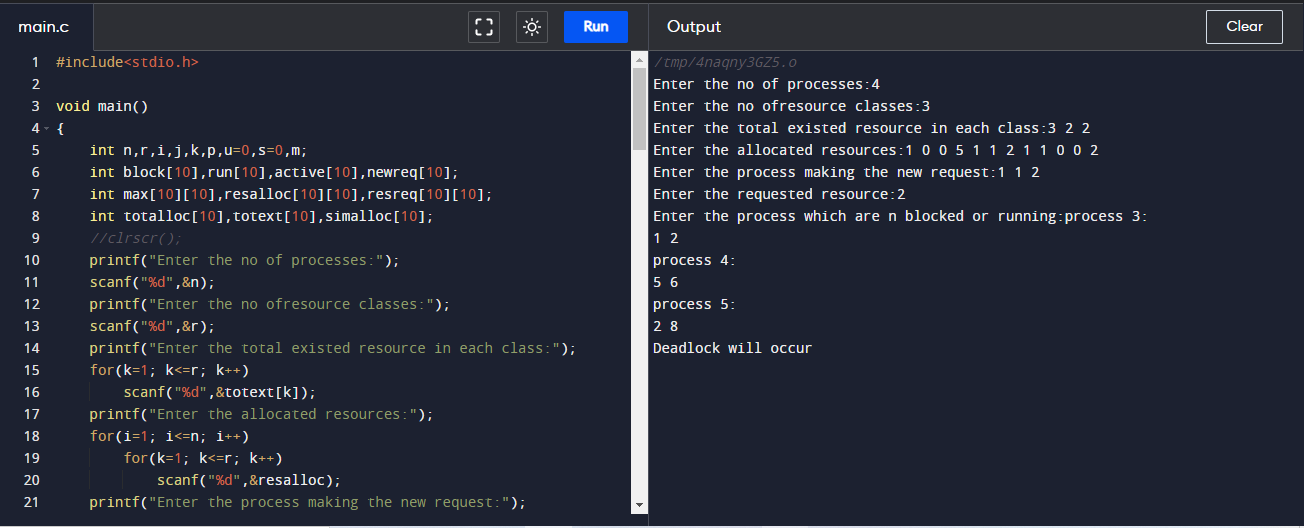
process 4:

5 6

process 5:

2 8

Deadlock will occur

****

RESULT:

Thus the deadlock avoidance concept by simulating Banker’s algorithm with C was successfully executed

EXERCISE 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

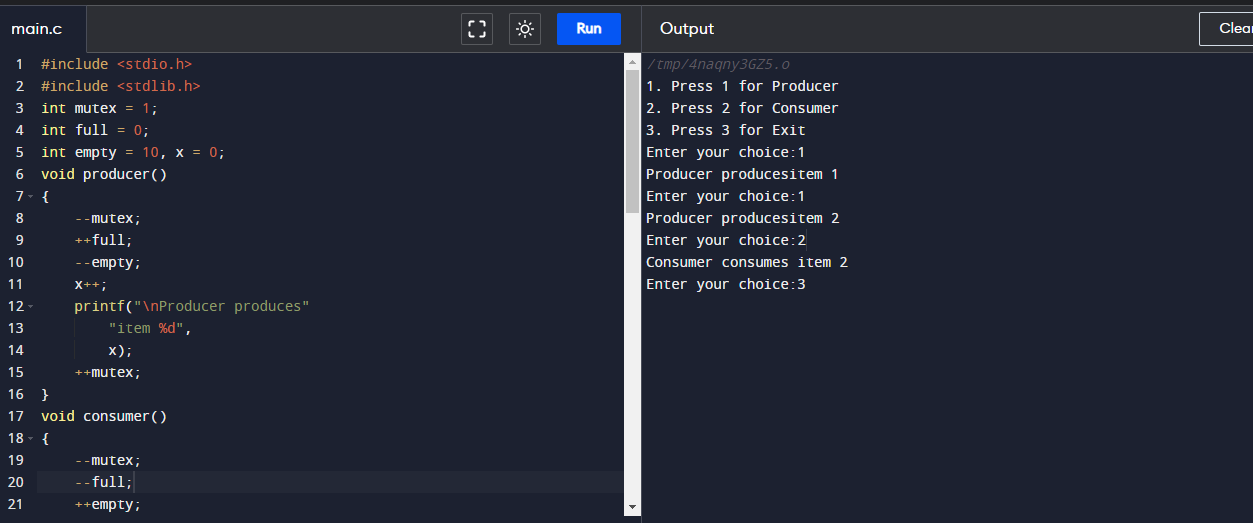
Enter your choice:1

Producer producesitem 2

Enter your choice:2

Consumer consumes item 2

Enter your choice:3



RESULT:

Thus Construction of a C program to simulate producer-consumer problem using semaphores was successfully executed.

EXERCISE 7-

AIM:

Construct a program to simulate the First in First Out paging technique of memory management. When a page must be replaced, the oldest page is chosen.

ALGORITHM:

STEP 1. Start to traverse the pages.

STEP 2. If the memory holds fewer pages, then the capacity else goes to step 5.

STEP 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.

STEP 4. If the current page is present in the memory, do nothing.

STEP 5. Else, pop the topmost page from the queue as it was inserted first.

STEP 6. Replace the topmost page with the current page from the string.

STEP7. Increment the page faults.

STEP 8. Stop

PROGRAM:

#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 Frame 1 \t Frame 2 \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("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}

OUTPUT:

Incoming Frame 1 Frame 2 Frame 3

4 4 - -

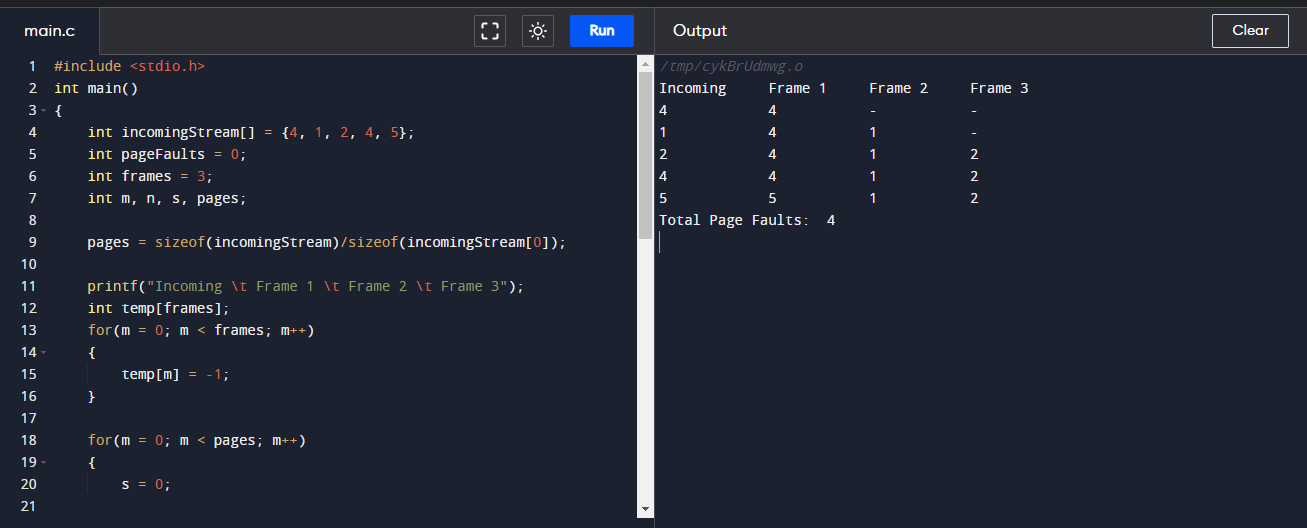
1 4 1 -

2 4 1 2

4 4 1 2

5 5 1 2

Total Page Faults: 4



RESULT:

Thus Construction of a C program to simulate the First in First Out paging technique of memory management was successfully completed.

EXPERIMENT 8-

AIM:

Construct a program to simulate the Least Recently Used paging technique of memory management. When a page must be replaced, the oldest page is chosen .

ALGORITHM:

STEP 1. Start the process

STEP 2. Declare the page size

STEP 3. Determine the number of pages to be inserted.

STEP 4. Get the value.

STEP 5. Declare the counter and stack value.

STEP 6. Choose the least recently used page by the counter value.

STEP 7. Stack them as per the selection.

STEP 8. Display the values.

STEP 9. Terminate the process.

PROGRAM:

#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 Frame1 \t Frame2 \t Frame3\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:

Page Frame1 Frame2 Frame3

1: 1

2: 1 2

3: 1 2 3

2: 1 2 3

1: 1 2 3

5: 1 2 5

2: 1 2 5

1: 1 2 5

6: 1 2 6

2: 1 2 6

5: 5 2 6

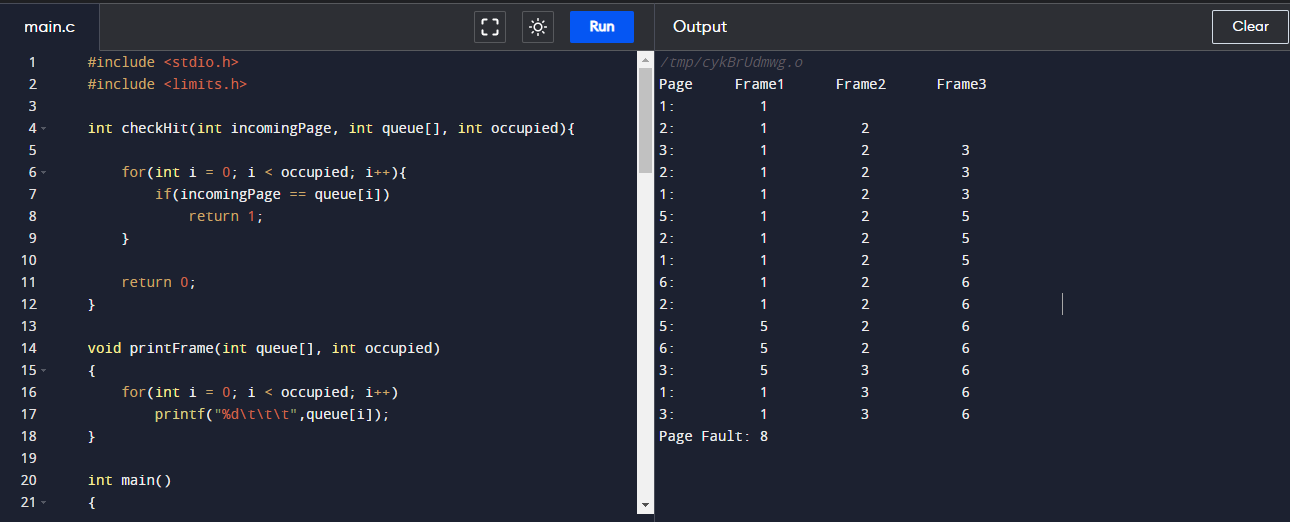
6: 5 2 6

3: 5 3 6

1: 1 3 6

3: 1 3 6

Page Fault: 8



RESULT:

Thus Construction a program to simulate the Least Recently Used paging technique of memory management was successfully completed.

EXPERIMENT 9-

AIM:

Construct a program to simulate the optimal paging technique of memory management. The operating system replaces the page that will not be used for the longest period of time in future.

ALGORITHM:

STEP 1:Push the first page in the stack as per the memory demand**.**

STEP 2: Push the second page as per the memory demand.

STEP 3: Push the third page until the memory is full**.**

STEP 4**:**As the queue is full, the page which is least recently used is popped.

STEP 5: Repeat step 4 until the page demand continues and until the processing is over.

STEP 6:Terminate the program.

PROGRAM:

#include <stdio.h>

int search(int key, int frame\_items[], int frame\_occupied)

{

for (int i = 0; i < frame\_occupied; i++)

if (frame\_items[i] == key)

return 1;

return 0;

}

void printOuterStructure(int max\_frames){

printf("Stream ");

for(int i = 0; i < max\_frames; i++)

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

}

void printCurrFrames(int item, int frame\_items[], int frame\_occupied, int max\_frames){

printf("\n%d \t\t", item);

for(int i = 0; i < max\_frames; i++){

if(i < frame\_occupied)

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

else

printf("- \t\t");

}

}

int predict(int ref\_str[], int frame\_items[], int refStrLen, int index, int frame\_occupied)

{

int result = -1, farthest = index;

for (int i = 0; i < frame\_occupied; i++) {

int j;

for (j = index; j < refStrLen; j++)

{

if (frame\_items[i] == ref\_str[j])

{

if (j > farthest) {

farthest = j;

result = i;

}

break;

}

}

if (j == refStrLen)

return i;

}

return (result == -1) ? 0 : result;

}

void optimalPage(int ref\_str[], int refStrLen, int frame\_items[], int max\_frames)

{

int frame\_occupied = 0;

printOuterStructure(max\_frames);

int hits = 0;

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

if (search(ref\_str[i], frame\_items, frame\_occupied)) {

hits++;

printCurrFrames(ref\_str[i], frame\_items, frame\_occupied, max\_frames);

continue;

}

if (frame\_occupied < max\_frames){

frame\_items[frame\_occupied] = ref\_str[i];

frame\_occupied++;

printCurrFrames(ref\_str[i], frame\_items, frame\_occupied, max\_frames);

}

else {

int pos = predict(ref\_str, frame\_items, refStrLen, i + 1, frame\_occupied);

frame\_items[pos] = ref\_str[i];

printCurrFrames(ref\_str[i], frame\_items, frame\_occupied, max\_frames);

}

}

printf("\n\nHits: %d\n", hits);

printf("Misses: %d", refStrLen - hits);

}

int main()

{

int ref\_str[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int refStrLen = sizeof(ref\_str) / sizeof(ref\_str[0]);

int max\_frames = 3;

int frame\_items[max\_frames];

optimalPage(ref\_str, refStrLen, frame\_items, max\_frames);

return 0;

}

OUTPUT:

stream Frame1 Frame2 Frame3

7 7 - -

0 7 0 -

1 7 0 1

2 2 0 1

0 2 0 1

3 2 0 3

0 2 0 3

4 2 4 3

2 2 4 3

3 2 4 3

0 2 0 3

3 2 0 3

2 2 0 3

1 2 0 1

2 2 0 1

0 2 0 1

1 2 0 1

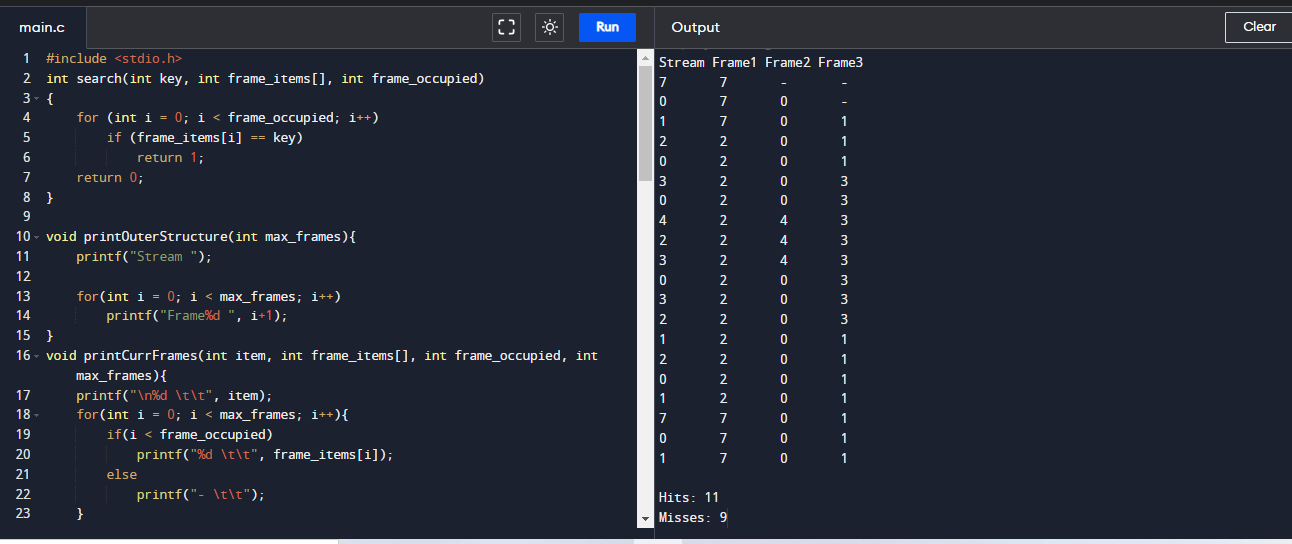
7 7 0 1

0 7 0 1

1 7 0 1

Hits: 11

Misses: 9



RESULT:

Thus Construction of a program to simulate the optimal paging technique of memory management was successfully completed.

EXERCISE 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: Gather information about the number of files.

STEP 3: Gather the memory requirement of each file.

STEP 4: Allocate the memory to the file in a sequential manner.

STEP 5: Select any random location from the available location.

STEP 6: Check if the location that is selected is free or not.

STEP 7: If the location is allocated set the flag = 1.

STEP 8: Print the file number, length, and the block allocated.

STEP 9: Gather information if more files have to be stored.

STEP 10: If yes, then go to STEP 2.

STEP 11: If no, Stop the program.

PROGRAM:

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

void recurse(int files[]){

int flag = 0, startBlock, len, j, k, ch;

printf("Enter the starting block and the length of the files: ");

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

for (j=startBlock; j<(startBlock+len); j++){

if (files[j] == 0)

flag++;

}

if(len == flag){

for (int k=startBlock; k<(startBlock+len); k++){

if (files[k] == 0){

files[k] = 1;

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

}

}

if (k != (startBlock+len-1))

printf("The file is allocated to the disk\n");

}

else

printf("The file is not allocated to the disk\n");

printf("Do you want to enter more files?\n");

printf("Press 1 for YES, 0 for NO: ");

scanf("%d", &ch);

if (ch == 1)

recurse(files);

else

exit(0);

return;

}

int main()

{

int files[50];

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

files[i]=0;

printf("Files Allocated are :\n");

recurse(files);

getch();

return 0;

}

OUTPUT:

Files Allocated are :

Enter the starting block and the length of the files: 14 3

14 1

15 1

16 1

The file is allocated to the disk

Do you want to enter more files?

Press 1 for YES, 0 for NO: 1

Enter the starting block and the length of the files: 15 8

The file is not allocated to the disk

Do you want to enter more files?

Press 1 for YES, 0 for NO: 1

Enter the starting block and the length of the files: 13 2

The file is not allocated to the disk

Do you want to enter more files?

Press 1 for YES, 0 for NO: 1

Enter the starting block and the length of the files: 45 56

The file is not allocated to the disk

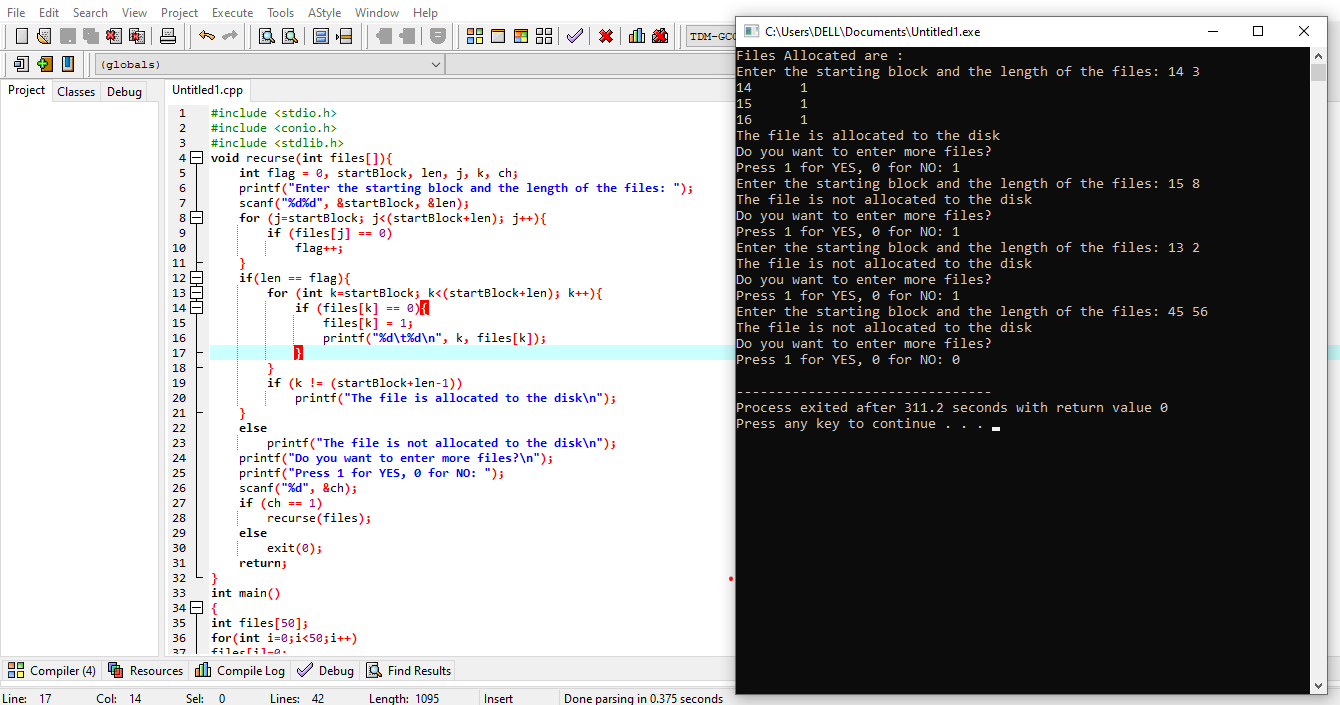
Do you want to enter more files?

Press 1 for YES, 0 for NO: 0

--------------------------------

Process exited after 311.2 seconds with return value 0

Press any key to continue . . .



RESULT:

Thus Designing a program to simulate the file allocation strategy was successfully completed.

EXERCISE 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 the program.

STEP 2: Get information about the number of files.

STEP 3: Get the memory requirement of each file.

STEP 4: Allocate the memory to the file by selecting random locations.

STEP 5: Check if the location that is selected is free or not.

STEP 6: If the location is allocated set the flag = 1, and if free set flag = 0.

STEP 7: Print the file number, length, and the block allocated.

STEP 8: Gather information if more files have to be stored.

STEP 9: If yes, then go to STEP 2.

STEP 10: If no, Stop the program.

PROGRAM:

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

int files[50], indexBlock[50], indBlock, n;

void recurse1();

void recurse2();

void recurse1(){

printf("Enter the index block: ");

scanf("%d", &indBlock);

if (files[indBlock] != 1){

printf("Enter the number of blocks and the number of files needed for the index %d on the disk: ", indBlock);

scanf("%d", &n);

}

else{

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

recurse1();

}

recurse2();

}

void recurse2(){

int ch;

int flag = 0;

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

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

if (files[indexBlock[i]] == 0)

flag++;

}

if (flag == n){

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

files[indexBlock[j]] = 1;

}

printf("Allocated\n");

printf("File Indexed\n");

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

printf("%d ------> %d : %d\n", indBlock, indexBlock[k], files[indexBlock[k]]);

}

}

else{

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

printf("Enter another indexed file\n");

recurse2();

}

printf("Do you want to enter more files?\n");

printf("Enter 1 for Yes, Enter 0 for No: ");

scanf("%d", &ch);

if (ch == 1)

recurse1();

else

exit(0);

return;

}

int main()

{

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

files[i]=0;

recurse1();

return 0;

}

OUTPUT:

Enter the index block: 3

Enter the number of blocks and the number of files needed for the index 3 on the disk: 1 2 3

Allocated

File Indexed

3 ------> 2 : 1

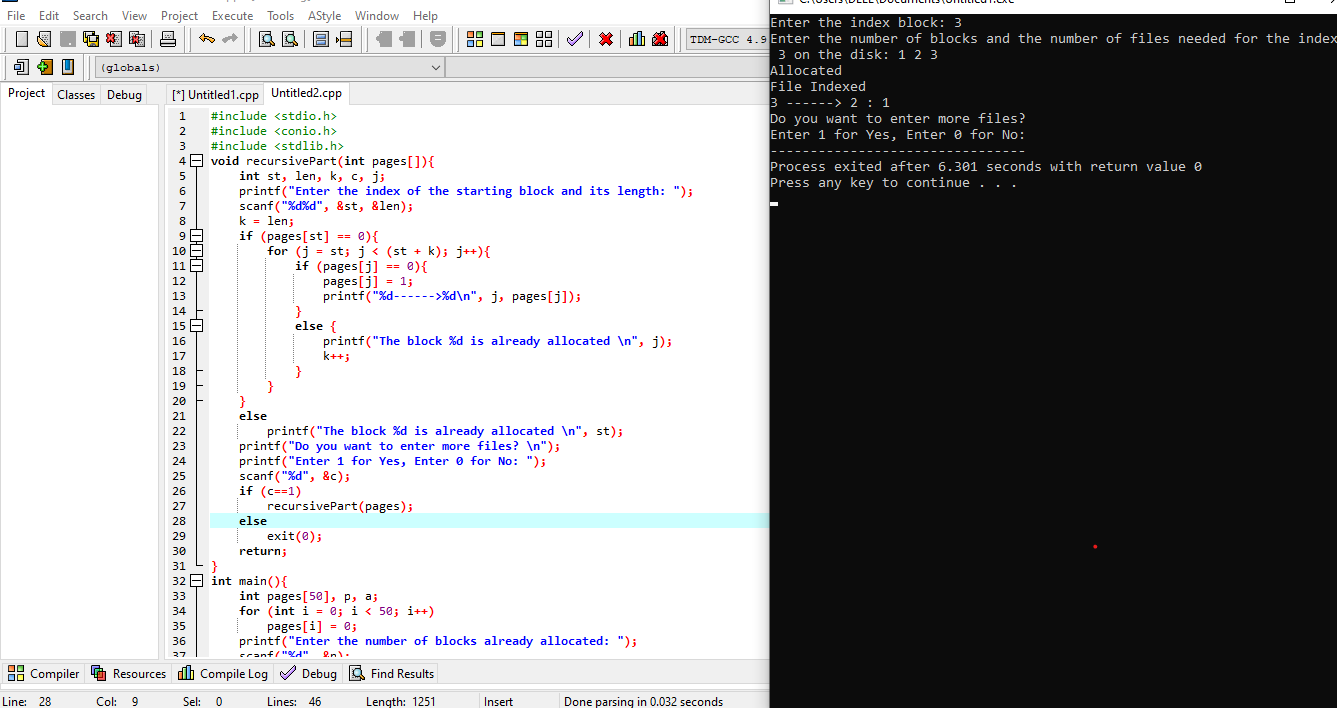
Do you want to enter more files?

Enter 1 for Yes, Enter 0 for No:

--------------------------------

Process exited after 6.301 seconds with return value 0

Press any key to continue . . .



RESULT:

Thus Designing a program to simulate the file allocation strategy was successfully executed.

EXERCISE 12:

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: Start the program.

STEP 2: Gather information about the number of files.

STEP 3: Allocate random locations to the files.

STEP 4: Check if the location that is selected is free or not.

STEP 5: If the location is free set the flag=0 a location is allocated set the flag = 1.

STEP 6: Print the file number, length, and the block allocated.

STEP 7: Gather information if more files have to be stored.

STEP 8: If yes, then go to STEP 2.

STEP 9: If no, Stop the program.

PROGRAM:

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

void recursivePart(int pages[]){

int st, len, k, c, j;

printf("Enter the index of the starting block and its length: ");

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

k = len;

if (pages[st] == 0){

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

if (pages[j] == 0){

pages[j] = 1;

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

}

else {

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

k++;

}

}

}

else

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

printf("Do you want to enter more files? \n");

printf("Enter 1 for Yes, Enter 0 for No: ");

scanf("%d", &c);

if (c==1)

recursivePart(pages);

else

exit(0);

return;

}

int main(){

int pages[50], p, a;

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

pages[i] = 0;

printf("Enter the number of blocks already allocated: ");

scanf("%d", &p);

printf("Enter the blocks already allocated: ");

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

scanf("%d", &a);

pages[a] = 1;

}

recursivePart(pages);

getch();

return 0;

}

OUTPUT:

Enter the number of blocks already allocated: 3

Enter the blocks already allocated: 1 3 5

Enter the index of the starting block and its length: 2 2

2------>1

The block 3 is already allocated

4------>1

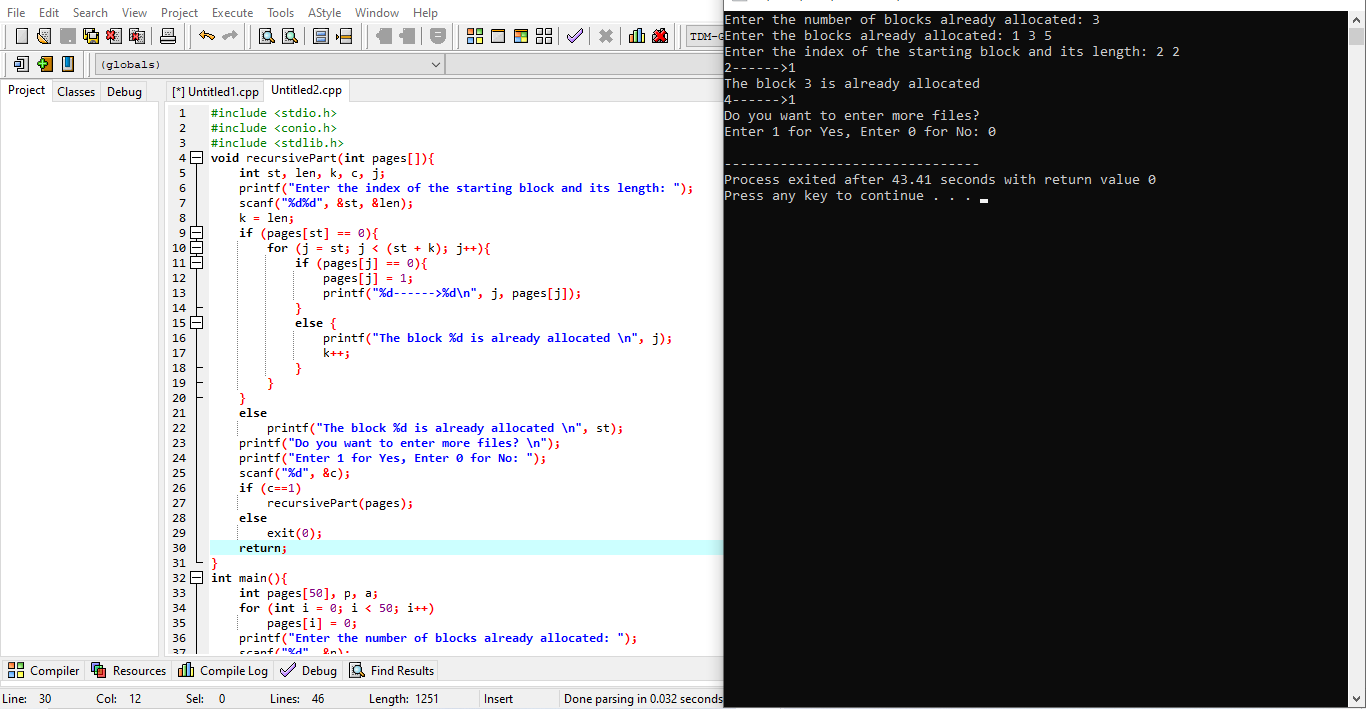
Do you want to enter more files?

Enter 1 for Yes, Enter 0 for No: 0

--------------------------------

Process exited after 43.41 seconds with return value 0

Press any key to continue . . .



RESULT:

Thus Designing a program to simulate the file allocation strategy was successfully completed.

EXERCISE 13:

AIM:

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

ALGORITHM:

 STEP 1: In function int waiting time (int proc [], int n, int burst time [], int wait time [])

* Set wait time [0] = 0
* Loop For i= 1 and I < n and i++
* Set wait time [i] = burst time[i-1] + wait time[i-1]
* End For

STEP 2: In function int turn around time ( int proc[], int n, int burst time[], int wait time[], int tat [])

* Loop For  i = 0 and i < n and i++
* Set tat[i] = burst time[i] + wait time[i]
* End For

STEP 3: In function int avg time( int proc[], int n, int burst time[])

* Declare and initialize wait time[n], tat[n], total wt= 0, total tat = 0;
* Call waiting time (proc, n, burst time, wait time)
* Call turn around time(proc, n, burst time, wait time, tat)
* Loop For  i=0 and i<n and i++
* Set total wt = total wt + wait time[i]
* Set total tat = total tat + tat[i]
* Print process number, burst time wait time and turnaround time
* End For
* Print "Average waiting time =i.e. total wt / n
* Print "Average turn around time = i.e. total tat / n

STEP 4: In int main ()

* Declare the input int proc [] = { 1, 2, 3}
* Declare and initialize n = size of proc / size of proc [0]
* Declare and initialize burst time[] = {10, 5, 8}
* Call avg time(proc, n, burst\_time)
* Stop

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):4

Enter Process Burst Time

P[1]:8

P[2]:4

P[3]:5

P[4]:6

Process Burst Time Waiting Time Turnaround Time

P[1] 8 0 8

P[2] 4 8 12

P[3] 5 12 17

P[4] 6 17 23

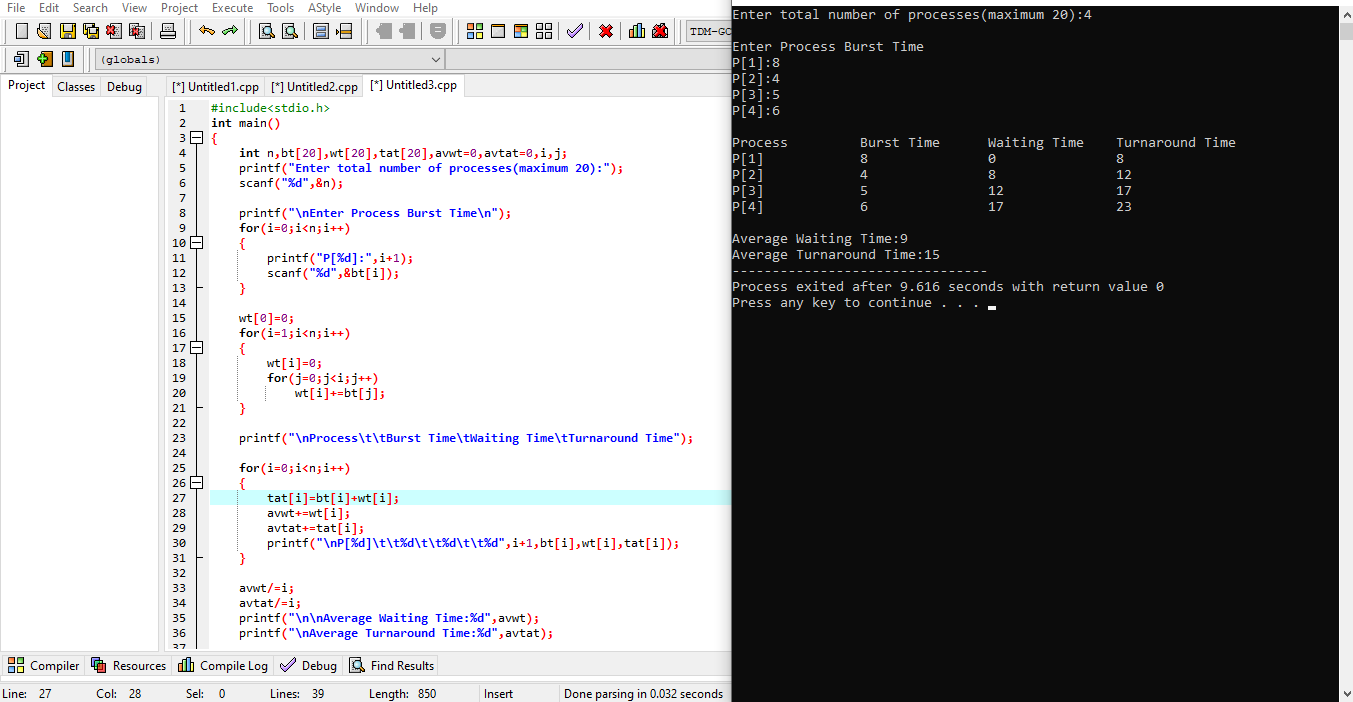
Average Waiting Time:9

Average Turnaround Time:15

--------------------------------

Process exited after 9.616 seconds with return value 0

Press any key to continue . . . .



RESULT:

Thus simulation of the First Come First Served disk scheduling algorithm is successfully completed.

EXPERIMENT 14:

AIM:

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

INTRODUCTION

In C, the data stored in a file can be accessed in the following ways:

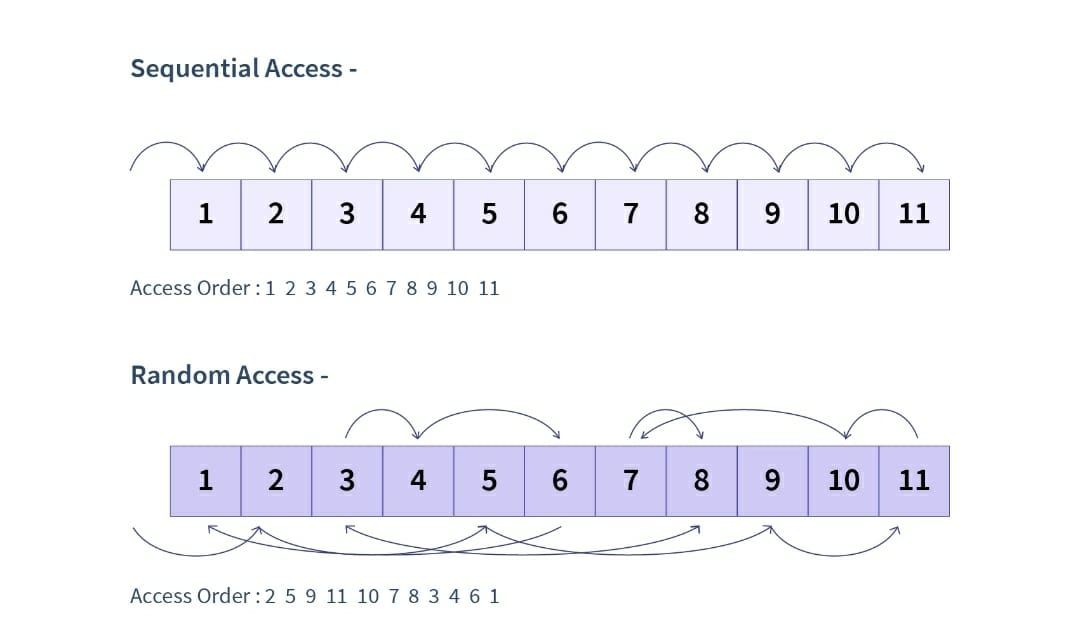
**SEQUENTIAL ACCESS:**

* If the file size is too huge, sequential access is not the best option for reading the record in the middle of the file.
* Random access to a file can be employed in this situation, enabling access to any record at any point in the file.
* We can imagine data in a random access file as songs on a compact disc or record; we can fast forward directly to any song we want without playing the other pieces.
* We can do so if we're playing the first song, the sixth song, the fourth song.
* This order has nothing to do with the songs' order initially recorded.
* Random file access sometimes takes more programming but rewards our effort with a more flexible file-access method.
* Thus, there are 3 functions which help in using the random access file in C:

fseek()

ftell()

rewind()



HOW TO USE THE F TELL () FUNCTION IN C HIGHLIGHTS :

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

ITS SYNTAX IS AS FOLLOWS:

ftell(FILE \*fp)

In C, the function ftell() is used to determine the location of the file pointer in the file relative to the file's beginning. ftell() has the following syntax:

pos = ftell(FILE \*fp);

Where, fp is a file pointer and pos holds the current position i.e., total bytes read (or written). For Example: If a file has 20 bytes of data and if the ftell() function returns 5 then, it means that 5 bytes has already been read (or written). Consider the below program to understand the ftell() function:

First, let us consider a file - Scaler.txt which contains the following data:

Scaler is amazing

Now let us see the code in C:

#include<stdio.h>

int main()

{

FILE \*fp;

fp = fopen("C:\\Users\\DELL\\Documents\\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)

{

OUTPUT:

Position of the pointer : 0

chingu is girl not .P

Position of the pointer : 23

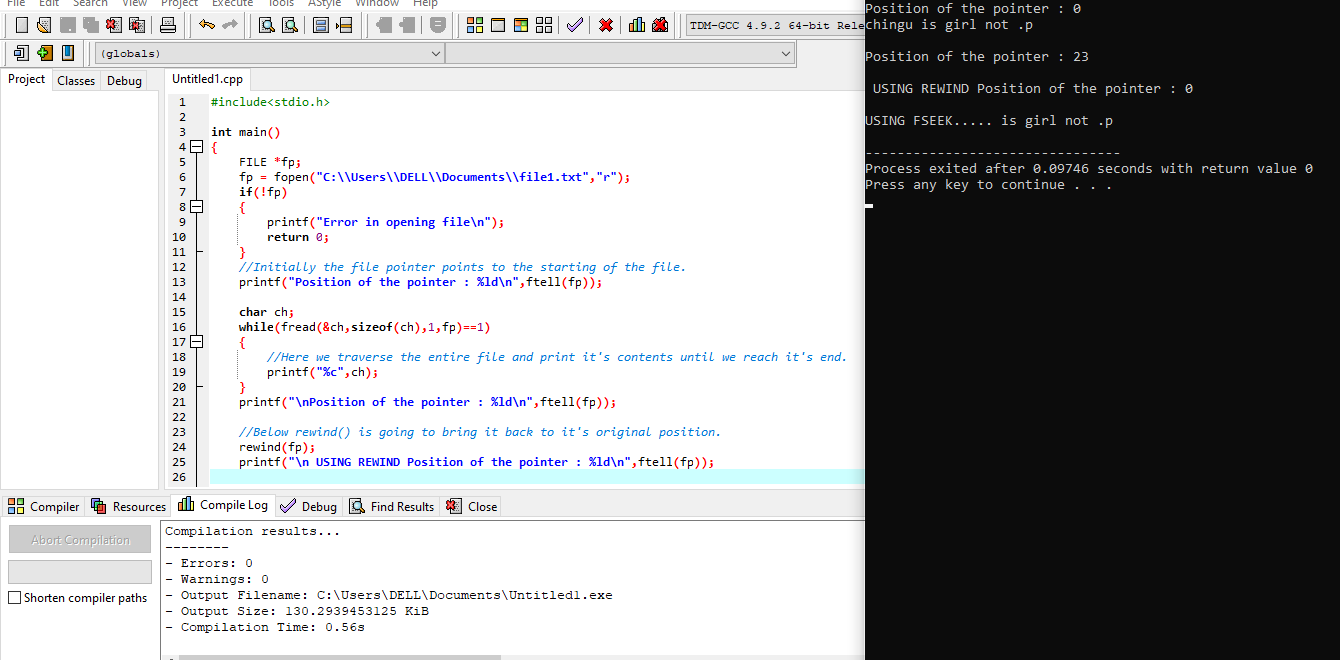
USING REWIND Position of the pointer : 0

USING FSEEK..... is girl not .p

--------------------------------

Process exited after 0.1006 seconds with return value 0

Press any key to continue . . .



HOW TO USE THE REWIND() FUNCTION IN C

HIGHLIGHTS:

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

Its syntax is as follows:

**rewind(FILE \*fp);**

The file pointer is moved to the beginning of the file using this function. It comes in handy when we need to update a file.

The following is the syntax:

**rewind(FILE \*fp);**

Here, fp is file pointer of type FILE. Consider the following program to understand the rewind() function:

#include<stdio.h>

int main()

{

FILE \*fp;

fp = fopen("C:\\Users\\DELL\\Documents\\file1.txt","r");

if(!fp)

{

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

return 0;

}

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

char ch;

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

{

printf("%c",ch);

}

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

rewind(fp);

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

fclose(fp);

return 0;

}

OUTPUT:

Position of the pointer : 0

chingu is girl not .p

Position of the pointer : 23

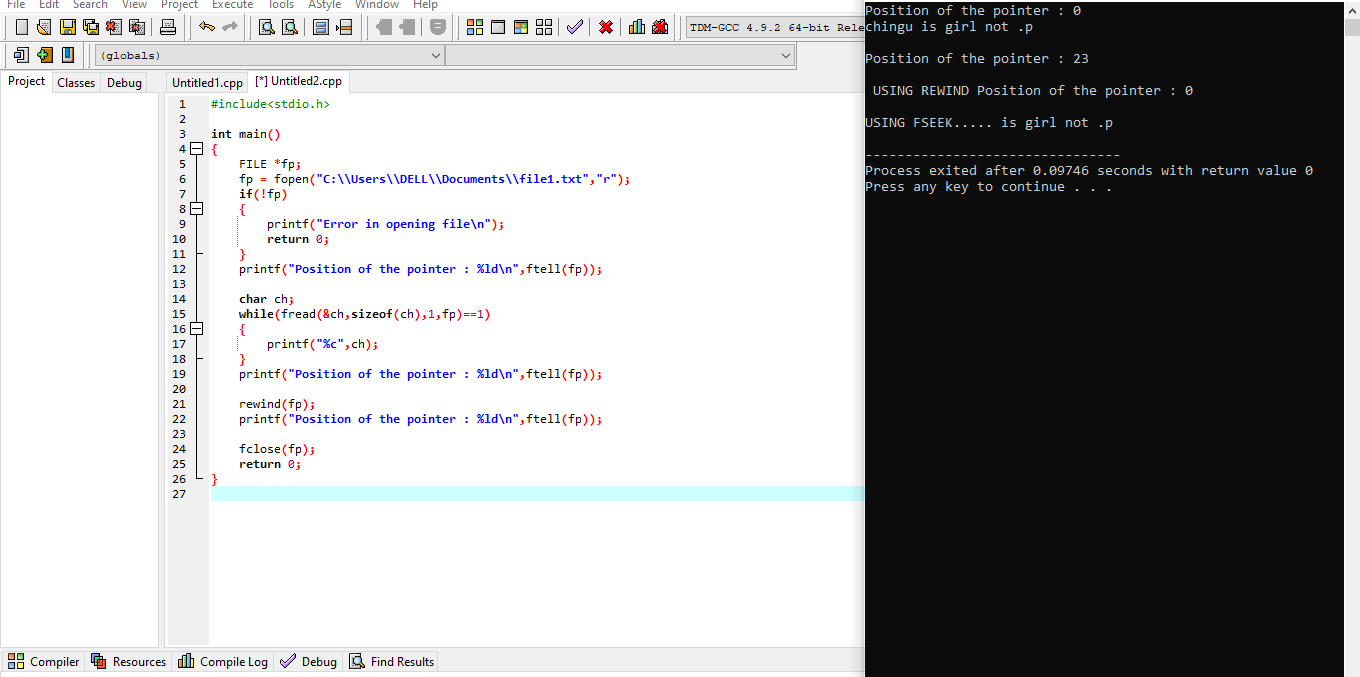
USING REWIND Position of the pointer : 0

USING FSEEK..... is girl not .p

--------------------------------

Process exited after 0.09746 seconds with return value 0

Press any key to continue . . .



HOW TO USE THE FSEEK() FUNCTION IN C

HIGHLIGHTS:

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

Its syntax is:

**int fseek(FILE \*fp, long displacement, int origin);**

To shift the file position to a specified place, use the fseek() function.

Syntax:

**int fseek(FILE \*fp, long displacement, int origin);**

Let us see the below program to understand the fseek() function:

#include<stdio.h>

int main()

{

FILE \*fp;

fp = fopen("C:\\Users\\DELL\\Documents\\file1.txt","r");

if(!fp)

{

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

return 0;

}

fseek(fp, 6, 0);

char ch;

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

{

printf("%c",ch);

}

fclose(fp);

return 0;

}

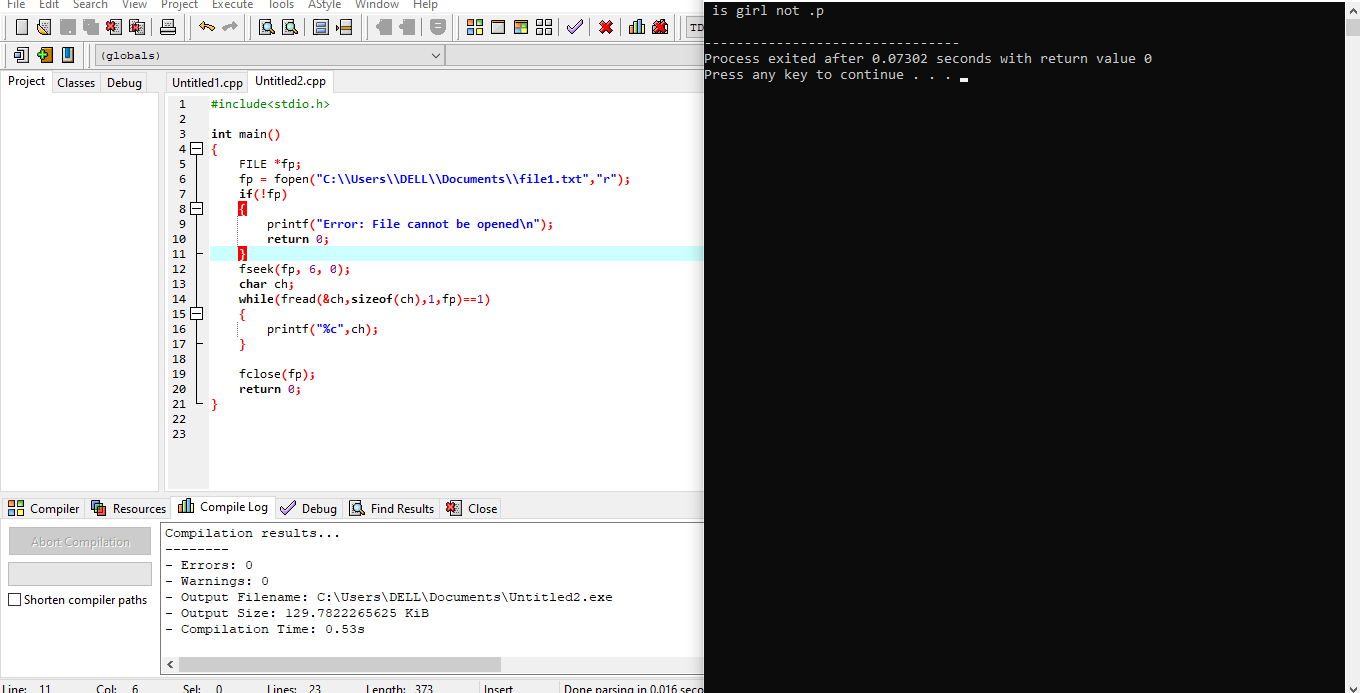
OUTPUT:

is girl not .p

--------------------------------

Process exited after 0.07302 seconds with return value 0

Press any key to continue . . .



EXERCISE 15:

AIM:

Study the features and deployment processes of Xen and VMware on Linux.

PROCEDURE:

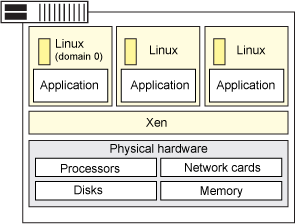
XEN:

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; EMC was later acquired by Dell Technologies in 2016. VMware bases its virtualization technologies on its bare-metal hypervisor ESX/ESXi in x86 architecture.

With VMware server virtualization, a hypervisor is installed on the physical server to allow for multiple virtual machines (VMs) to run on the same physical server. Each VM can run its own operating system (OS), which means multiple OSes can run on one physical server. All the VMs on the same physical server share resources, such as networking and RAM. In 2019, VMware added support to its hypervisor to run containerized workloads in a Kubernetes cluster in a similar way. These types of workloads can be managed by the infrastructure team in the same way as virtual machines and the DevOps teams can deploy containers as they were used to.

Diane Greene, Scott Devine, Mendel Rosenblum, Edward Wang and Edouard Bugnion founded VMware, which launched its first product -- VMware Workstation -- in 1999. The company released its second product, VMware ESX in 2001.

VMware's current CEO is Patrick Gelsinger, appointed in 2012.

VMWARE PRODUCTS

VMware products include virtualization, networking and security management tools, software-defined data center software and storage software.

Data center and cloud infrastructure

VMware vSphere is VMware's suite of virtualization products. VMware vSphere, known as VMware Infrastructure prior to 2009, includes the following:

• ESXi

• vCenter Server

• vSphere Client

• vMotion

As of April 2018, the most current version is vSphere 6.7, which is available in three editions: Standard, Enterprise Plus and Platinum. There are also two three-server kits targeted toward small and medium-sized businesses named vSphere Essentials and Essentials Plus.

With VMware Cloud on AWS, customers can run a cluster of vSphere hosts with vSAN and NSX in an Amazon data center and run their workloads there while in the meantime manage them with their well-known VMware tools and skills.

NETWORKING AND SECURITY

VMware NSX is a virtual networking and security software offering created when VMware acquired Nicera in 2012. NSX allows an admin to virtualize network components, enabling them to develop, deploy and configure virtual networks and switches through software rather than hardware. A software layer sits on top of the hypervisor to allow an administrator to divide a physical network into multiple virtual networks.

With the latest release of the product, NSX-T Data Center, network virtualization can be added to both ESXi and KVM as hypervisors, as well as to bare-metal servers. Also containerized workloads in a Kubernetes cluster can be virtualized and protected. NSX-T Data Center also offers Network Function Virtualization, with which functions such as a firewall, load balancer and VPN, can be run in the virtualization software stack.

VMware vRealize Network Insight is a network operations management tool that enables an admin to plan microsegmentation and check on the health of VMware NSX. VRealize Network Insight relies on technology from VMware's acquisition of Arkin in 2016. VRealize Network Insight collects information from the NSX Manager. It also displays errors in its user interface, which helps troubleshoot an NSX environment.