**EXERCISE:3**

**SIMULATE MULTI THREADED CONCEPT USING THE PTHREADS API**

#include<pthread.h>

#include<stdio.h>

int sum;

/\* This ‘sum’ is shared by the thread(s) \*/

void \*runner(void \*param);

/\* threads call this function \*/

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

pthread t tid;

/\* the thread identifier \*/

pthread attr t attr;

/\* set of thread attributes \*/

if (argc != 2){

fprintf(stderr,"usage: a.out");

return -1;

}

if (atoi(argv[1]) < 0){

fprintf(stderr,"%d must be >= 0",atoi(argv[1])); return -1;

}

/\* get the default attributes \*/

pthread attr init(&attr);

/\* create the thread \*/

pthread create(&tid,&attr,runner,argv[1]);

/\* wait for the thread to exit \*/

pthread join(tid,NULL);

printf("sum = %d",sum);

}

/\* The thread will begin handle in this function \*/

void \*runner(void \*param){

int i, upper = atoi(param);

sum = 0;

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

sum += i;

pthread exit(0);

}

1.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h> //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>

// A normal C function that is executed as a thread

// when its name is specified in pthread\_create()

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);

}

2.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h> //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>

// A normal C function that is executed as a thread

// when its name is specified in pthread\_create()

void \*myThreadFun(void \*vargp)

{

sleep(1);

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

return NULL;

}

void \*mysecfun(void \*vargp)

{

sleep(2);

printf("this is a second thread");

return NULL;

}

int main()

{

pthread\_t thread\_id1,thread\_id2;

printf("Before Thread\n");

pthread\_create(&thread\_id1, NULL, myThreadFun, NULL);

pthread\_create(&thread\_id2,NULL, mysecfun, NULL);

pthread\_join(thread\_id1, NULL);

pthread\_join(thread\_id2, NULL);

printf("threads are executed");

}

**EXERCISE: 4**

**CPU Scheduling algorithms:**

1. **FCFS Algorithm**

// C program for implementation of FCFS

// scheduling

#include<stdio.h>

// Function to find the waiting time for all

// processes

void findWaitingTime(int processes[], int n,

                          int bt[], int wt[])

{

    // waiting time for first process is 0

    wt[0] = 0;

    // calculating waiting time

    for (int  i = 1; i < n ; i++ )

        wt[i] =  bt[i-1] + wt[i-1] ;

}

// Function to calculate turn around time

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];

}

//Function to calculate average time

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

{

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

    //Function to find waiting time of all processes

    findWaitingTime(processes, n, bt, wt);

    //Function to find turn around time for all processes

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

    //Display processes along with all details

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

    // Calculate total waiting time and total turn

    // around time

    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] );

    }

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

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

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

    printf("\n");

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

}

// Driver code

int main()

{

    //process id's

    int processes[] = { 1, 2, 3};

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

    //Burst time of all processes

    int  burst\_time[] = {10, 5, 8};

    findavgTime(processes, n,  burst\_time);

    return 0;

}

**Output:**

Processes Burst time Waiting time Turn around time //The Output is Wrong please correct it

1 10 0 10

2 5 10 15

3 8 15 23

Average waiting time = 8.33333

Average turn around time = 16

1. **SJF Algorithm:**

/ \* C Program to Implement SJF Scheduling \*/

#include<stdio.h>

int main()

{

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

}

//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;

//finding the waiting time of all the processes

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

{

wt[i]=0;

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

//individual WT by adding BT of all previous completed processes

wt[i]+=bt[j];

//total waiting time

total+=wt[i];

}

//average waiting time

avg\_wt=(float)total/n;

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

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

{

//turnaround time of individual processes

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

//total turnaround time

totalT+=tat[i];

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

}

//average turnaround time

avg\_tat=(float)totalT/n;

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

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

}

**Output:**

Enter number of process:4

Enter Burst Time:

p1:5

p2:4

p3:12

p4:7

Process Burst Time Waiting Time Turnaround Time

p2 4 0 4

p1 5 4 9

p4 7 9 16

p3 12 16 28

Average Waiting Time=7.250000

Average Turnaround Time=14.250000

**3. ROUND ROBIN:**

/\* Round Robin Scheduling Program in C \*/

#include<stdio.h>

int main()

{

//Input no of processed

int n;

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

scanf("%d", &n);

int wait\_time = 0, ta\_time = 0, arr\_time[n], burst\_time[n], temp\_burst\_time[n];

int x = n;

//Input details of processes

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

{

printf("Enter Details of Process %d \n", i + 1);

printf("Arrival Time: ");

scanf("%d", &arr\_time[i]);

printf("Burst Time: ");

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

temp\_burst\_time[i] = burst\_time[i];

}

//Input time slot

int time\_slot;

printf("Enter Time Slot:");

scanf("%d", &time\_slot);

//Total indicates total time

//counter indicates which process is executed

int total = 0, counter = 0,i;

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

for(total=0, i = 0; x!=0; )

{

// define the conditions

if(temp\_burst\_time[i] <= time\_slot && temp\_burst\_time[i] > 0)

{

total = total + temp\_burst\_time[i];

temp\_burst\_time[i] = 0;

counter=1;

}

else if(temp\_burst\_time[i] > 0)

{

temp\_burst\_time[i] = temp\_burst\_time[i] - time\_slot;

total += time\_slot;

}

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

{

x--; //decrement the process no.

printf("\nProcess No %d \t\t %d\t\t\t\t %d\t\t\t %d", i+1, burst\_time[i],

total-arr\_time[i], total-arr\_time[i]-burst\_time[i]);

wait\_time = wait\_time+total-arr\_time[i]-burst\_time[i];

ta\_time += total -arr\_time[i];

counter =0;

}

if(i==n-1)

{

i=0;

}

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

{

i++;

}

else

{

i=0;

}

}

float average\_wait\_time = wait\_time \* 1.0 / n;

float average\_turnaround\_time = ta\_time \* 1.0 / n;

printf("\nAverage Waiting Time:%f", average\_wait\_time);

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

return 0;

}

**OUTPUT:**

Enter Total Number of Processes:3

Enter Details of Process 1

Arrival Time: 7

Burst Time: 8

Enter Details of Process 2

Arrival Time: 2

Burst Time: 5

Enter Details of Process 3

Arrival Time: 5

Burst Time: 7

8

7

Enter Time Slot:7

8

Process ID Burst Time Turnaround Time Waiting Time

Process No 2 5 10 5

Process No 3 7 14 7

Process No 1 8 13 5

Average Waiting Time:5.666667

Avg Turnaround Time:12.333333dash: 3: 8: not found

**PRIORITY ALGORITHM:**

/\* C program to implement priority scheduling \*/

#include <stdio.h>

//Function to swap two variables

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

{

int temp=\*a;

\*a=\*b;

\*b=temp;

}

int main()

{

int n;

printf("Enter Number of Processes: ");

scanf("%d",&n);

// b is array for burst time, p for priority and index for process id

int b[n],p[n],index[n];

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

{

printf("Enter Burst Time and Priority Value for Process %d: ",i+1);

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

index[i]=i+1;

}

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

{

int a=p[i],m=i;

//Finding out highest priority element and placing it at its desired position

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

{

if(p[j] > a)

{

a=p[j];

m=j;

}

}

//Swapping processes

swap(&p[i], &p[m]);

swap(&b[i], &b[m]);

swap(&index[i],&index[m]);

}

// T stores the starting time of process

int t=0;

//Printing scheduled process

printf("Order of process Execution is\n");

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

{

printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);

t+=b[i];

}

printf("\n");

printf("Process Id Burst Time Wait Time TurnAround Time\n");

int wait\_time=0;

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

{

printf("P%d %d %d %d\n",index[i],b[i],wait\_time,wait\_time + b[i]);

wait\_time += b[i];

}

return 0;

}

**OUTPUT:**

Enter Number of Processes: 4

Enter Burst Time and Priority Value for Process 1: 10 5

Enter Burst Time and Priority Value for Process 2: 8 6

Enter Burst Time and Priority Value for Process 3: 7 4

Enter Burst Time and Priority Value for Process 4: 9 2

Order of process Execution is

P2 is executed from 0 to 8

P1 is executed from 8 to 18

P3 is executed from 18 to 25

P4 is executed from 25 to 34

Process Id Burst Time Wait Time TurnAround Time

P2 8 0 8

P1 10 8 18

P3 7 18 25

P4 9 25 34

**EXERCISE:9**

**6 .AIM: To Implement a program Synchronization on Producer-Consumer Problem.**

**DESCRIPTION:** The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.The producer consumer problem can be resolved using semaphores.

**Source code**

#include<stdio.h>

#include<unistd.h>

#include<pthread.h>

#define buffsize 1000

struct

{

pthread\_mutex\_tmutex;

int buff[buffsize];

intpcount;

intccount;

}shared={PTHREAD\_MUTEX\_INITIALIZER};

void \*produce(void \*arg);

void \*consume(void \*arg);

int main()

{

intpthreads,cthreads,i;

printf("\nEnter the no.of Producers:");

scanf("%d",&pthreads);

printf("\nEnter the no.of Consumers:");

scanf("%d",&cthreads);

intparr[pthreads],carr[cthreads];

pthread\_t producer[pthreads],consumer[cthreads];

pthread\_setconcurrency(pthreads+cthreads);

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

{

parr[i]=0;

pthread\_create(&producer[i],NULL,produce,&parr[i]);

}

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

{

carr[i]=0;

pthread\_create(&consumer[i],NULL,consume,&carr[i]);

}

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

{

pthread\_join(producer[i],NULL);

printf("\nThe Producer(%d) produced:[%d] Items",i,parr[i]);

sleep(1);

}

printf("\n");

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

{

pthread\_join(consumer[i],NULL);

printf("\nThe Consumer(%d) Consumed:[%d] Items",i,carr[i]);

sleep(1);

}

return 0;

}

void \*produce(void \*arg)

{

while(1)

{

pthread\_mutex\_lock(&shared.mutex);

if(shared.pcount<buffsize)

{

shared.pcount++;

\*((int \*)arg)+=1;

pthread\_mutex\_unlock(&shared.mutex);

}

else

{

pthread\_mutex\_unlock(&shared.mutex);

return NULL;

}

}

}

void \*consume(void \*arg)

{

while(1)

{

pthread\_mutex\_lock(&shared.mutex);

if(shared.ccount<shared.pcount)

{

shared.ccount++;

\*((int \*)arg)+=1;

pthread\_mutex\_unlock(&shared.mutex);

}

else

{

pthread\_mutex\_unlock(&shared.mutex);

if(shared.ccount>=buffsize)

return NULL;

}

}

}

No of producers:4

No of consumers:3

**EXERCISE:10**

**5 a) Binary semaphore using mutex**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex;

void\* thread(void\* arg)

{

//wait

sem\_wait(&mutex);

printf("\nEntered thread\n");

//critical section

sleep(4);

//signal

printf("\n Exit thread\n");

sem\_post(&mutex);

}

int main()

{

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

pthread\_t t1,t2;

pthread\_create(&t1,NULL,thread,NULL);

sleep(2);

pthread\_create(&t2,NULL,thread,NULL);

pthread\_join(t1,NULL);

pthread\_join(t2,NULL);

sem\_destroy(&mutex);

return 0;

}

**5a) Binary semaphore using wait**

**DESCRIPTION:** a semaphore is a variable or abstract data type used to control access to a common resource by multiple [processes](https://en.wikipedia.org/wiki/Process_(computing)) in a [concurrent](https://en.wikipedia.org/wiki/Concurrent_computing) system such as a [multitasking](https://en.wikipedia.org/wiki/Computer_multitasking) operating system. A semaphore is simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multi processing environment. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores and are used to implement [locks](https://en.wikipedia.org/wiki/Lock_(computer_science)).

#include<pthread.h>

#include<stdio.h>

#include<semaphore.h>

#include<unistd.h>

void \*fun1();

void \*fun2();

int shared=1;

sem\_t s;

int main()

{

sem\_init(&s,0,1);

pthread\_t thread1,thread2;

pthread\_create(&thread1,NULL,fun1,NULL);

pthread\_create(&thread2,NULL,fun2,NULL);

pthread\_join(thread1,NULL);

pthread\_join(thread2,NULL);

printf("final value of shsred is %d\n",shared);

}

void \*fun1()

{

int x;

sem\_wait(&s);

x=shared;

printf("thread1 reads the value as %d\n",x);

x++;

printf("local updation by thread1: %d\n",x);

sleep(1);

shared=x;

printf("value of shared variable updated by thread1 is %d\n",shared);

sem\_post(&s);

}

void \*fun2()

{

int y;

sem\_wait(&s);

y=shared;

printf("thread2 reads the value as %d\n",y);

y--;

printf("local updation by thread2: %d\n",y);

sleep(1);

shared=y;

printf("value of shared variable updated by thread2 is %d\n",shared);

sem\_post(&s);

}

5.(b) AIM: To Implement a program on Counter Semaphore.

**DESCRIPTION**: a semaphore is a variable or abstract data type used to control access to a

common resource by multiple processes in a concurrent system such as a multitasking operating

system. A semaphore is simply a variable. This variable is used to solve critical section problems

and to achieve process synchronization in the multi processing environment. A trivial

semaphore is a plain variable that is changed (for example, incremented or decremented, or

toggled) depending on programmer-defined conditions. Semaphores which allow an arbitrary

resource count are called counting semaphores.

#include<semaphore.h>

#include<pthread.h>

#include<stdio.h>

pthread\_mutex\_t mutex;

sem\_t empty,full;

int size=05,buffer[100],count;

void \*consumer(void \*i)

{

int num=(int)i;

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

num=buffer[count-1];

count--;

printf("\nConsumed %d",num);

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

}

void \*producer(void \*i)

{

int num=(int)i;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[count]=num;

count++;

printf("\nProduced %d",buffer[count-1]);

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

}

int main()

{

pthread\_t tid[10],tid1[10];

pthread\_mutex\_init(&mutex,NULL);

int nop=5;

int noc=5;

int i,j;

sem\_init(&empty,0,size);

sem\_init(&full,0,0);

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

pthread\_create(&tid[i],NULL,producer,(void\*)i+1);

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

pthread\_join(tid[j],NULL);

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

pthread\_create(&tid1[i],NULL,consumer,(void\*)i+1);

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

pthread\_join(tid1[j],NULL);

}

/\*\*\*OUTPUT

Produced 2

Produced 4

Produced 1

Produced 5

Produced 3

Consumed 3

Consumed 5

Consumed 1

Consumed 4

Consumed 2 \*/

**8**).Aim Write a C program to simulate the concept of Dining-Philosophers problem.

**DESCRIPTION:** The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock

**Source code**

int tph, philname[20], status[20], howhung, hu[20],

cho; main()

{

int i;

printf("\n\nDINING PHILOSOPHER PROBLEM");

printf("\nEnter the total no. ofphilosophers: ");

scanf("%d",&tph);

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

{

philname[i] =(i+1);

status[i]=1;

}

printf("How many are hungry :");

scanf("%d", &howhung);

if(howhung==tph)

{

printf("\nAll are hungry..\nDead lock stage willoccur");

printf("\nExiting..");

}

else

{

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

{

printf("Enter philosopher %d position:",(i+1));

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

status[hu[i]]=2;

}

do

{

printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter yourchoice:");

scanf("%d", &cho);

switch(cho)

{

case 1: one();

break;

case 2: two();

break;

case 3: exit(0);

default: printf("\nInvalid option..");

}

}while(1);

}

}

one()

{

int pos=0, x, i;

printf("\nAllow one philosopher to eat at any time\n");

for(i=0;i<howhung; i++, pos++)

{

printf("\nP %d is granted to eat", philname[hu[pos]]);

for(x=pos;x<howhung;x++)

printf("\nP %d is waiting", philname[hu[x]]);

}

}

two()

{

int i, j, s=0, t, r, x;

printf("\n Allow two philosophers to eat at same time\n");

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

{

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

{

if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)

{

printf("\n\ncombination %d \n", (s+1));

t=hu[i];

r=hu[j];

s++;

printf("\nP %d and P %d are granted to eat", philname[hu[i]], philname[hu[j]]);

for(x=0;x<howhung;x++)

{

if((hu[x]!=t)&&(hu[x]!=r))

printf("\nP %d is waiting", philname[hu[x]]);

}

}

}

}

}

Exercise:11

9 a) DEAD LOCK AVOIDANCE

AIM: To Simulate bankers algorithm for Dead Lock Avoidance (Banker‘s Algorithm)

DESCRIPTION: Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the userrequest a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources. Data structures n-Number of process, m-number of resource types. Available: Available[j]=k, k – instance of resource type Rj is available. Max: If max[i, j]=k, Pi may request at most k instances resource Rj. Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj, Need[I, j]=Max[I, j]- Allocation[I, j];

**Safety Algorithm**

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False. 2. Find an i such that both Finish[i] =False Need<=Work If no such I exists go to step 4.

3. work= work + Allocation, Finish[i] =True;

4. if Finish[1]=True for all I, then the system is in safe state.

**Resource request algorithm**

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.

2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

Available=Available-Request I;

Allocation I=Allocation +Request I;

Need i=Need i- Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state isrestored.

**Source code**

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

int alloc[10][10],max[10][10];

int avail[10],work[10],total[10];

int i,j,k,n,need[10][10];

int m;

int count=0,c=0;

char finish[10];

printf("Enter the no. of processes and resources:");

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

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

finish[i]='n';

printf("Enter the claim matrix:\n");

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

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

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

printf("Enter the allocation matrix:\n");

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

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

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

printf("Resource vector:");

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

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

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

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

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

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

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

work[i]=avail[i];

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

work[j]=total[j]-work[j];

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

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

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

A:

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

{

c=0;

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

if((need[i][j]<=work[j])&&(finish[i]=='n'))

c++;

if(c==m)

{

printf("All the resources can be allocated to Process %d", i+1);

printf("\n\nAvailable resources are:");

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

{

work[k]+=alloc[i][k];

printf("%4d",work[k]);

}

printf("\n");

finish[i]='y';

printf("\nProcess %d executed?:%c \n",i+1,finish[i]);

count++;

}

}

if(count!=n)

goto A;

else

printf("\n System is in safe mode");

printf("\n The given state is safe state");

getch();

}

OUTPUT Enter the no. of processes and resources: 4 3

Enter the claim matrix:

3 2 2

6 1 3

3 1 4

4 2 2

Enter the allocation matrix:

1 0 0

6 1 2

2 1 1

0 0 2

Resource vector:9 3 6

All the resources can be allocated to Process 2

Available resources are: 6 2 3

Process 2 executed?:y

All the resources can be allocated to Process 3

Available resources are: 8 3 4

Process 3 executed?:y

All the resources can be allocated to Process 4

Available resources are: 8 3 6 Process 4 executed?:y

All the resources can be allocated to Process 1

Available resources are: 9 3 6 Process 1 executed?:y

System is in safe mode The given state is safe state

9 b)DEAD LOCKPREVENTION

AIM: To implement deadlock prevention technique Banker‘s Algorithm:

**DESCRIPTION**: Data structures n-Number of process, m-number of resource types. Available: Available[j]=k, k – instance of resource type Rj is available. Max: If max[i, j]=k, Pi may request at most k instances resource Rj. Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj,

Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm Work and Finish be the vector of length m and n respectively,

Work=Available and Finish[i] =False. Find an i such that both Finish[i] =False Need<=Work If no such I exists go to step 4. 5. work=work+Allocation, Finish[i] =True; if Finish[1]=True for all I, then the system is in safe state

**SOURCE CODE**

#include<stdio.h>

#include<conio.h>

void main()

{

char job[10][10];

int time[10],avail,tem[10],temp[10]; int safe[10];

int ind=1,i,j,q,n,t;

printf("Enter no of jobs: ");

scanf("%d",&n);

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

{

printf("Enter name and time: ");

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

}

printf("Enter the available resources:");

scanf("%d",&avail);

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

{

temp[i]=time[i];

tem[i]=i;

}

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

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

{

if(temp[i]>temp[j])

{

t=temp[i];

temp[i]=temp[j];

temp[j]=t; t=tem[i];

tem[i]=tem[j];

tem[j]=t;

}

}

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

{

q=tem[i];

if(time[q]<=avail)

{

safe[ind]=tem[i];

avail=avail-tem[q];

printf("%s",job[safe[ind]]);

ind++;

}

else

{

printf("No safe sequence\n");

}

}

printf("Safe sequence is:");

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

printf("%s %d\n",job[safe[i]],time[safe[i]]);

getch();

}

OUTPUT: Enter no of jobs:4

Enter name and time: A 1

Enter name and time: B 4

Enter name and time: C 2

Enter name and time: D 3

Enter the available resources: 20

Safe sequence is:

A 1, C 2, D 3, B 4.

AIM:To write a C program to implement Deadlock Detection algorithm

**DESCRIPTION**: Deadlock Detection is an important task of OS. As the OS doesn’t take many precautionary means to avoid it. The OS periodically checks if there is any existing deadlock in the system and take measures to remove the deadlocks.

#include<stdio.h>

void main()

{

int found,flag,l,p[4][5],tp,c[4][5],i,j,k=1,m[5],r[5],a[5],temp[5],sum=0;

printf("enter total no of processes: \n");

scanf("%d",&tp);

printf("enter clain matrix: \n");

for(i=0;i<4;i++)

{

for(j=0;j<5;j++)

{

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

}

}

printf("enter allocation matrix: \n");

for(i=0;i<4;i++)

{

for(j=0;j<5;j++)

{

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

}

}

printf("enter resource vector: \n");

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

{

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

}

printf("enter availability vector: \n");

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

{

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

temp[i]=a[i];

}

for(i=0;i<4;i++)

{

sum=0;

for(j=0;j<5;j++)

{

sum+=p[i][j];

}

if(sum==0)

{

m[k]=i;

k++;

}

}

for(i=0;i<4;i++)

{

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

{

if(i!=m[l])

{

flag=1;

for(j=0;j<5;j++)

{

if(c[i][j]>temp[j])

{

flag=0;

break;

}

}

}

}

if(flag==1)

{

m[k]=i;

k++;

for(j=0;j<5;j++)

temp[j]+=p[i][j];

}

}

printf("deadlock causing processes are: \n");

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

{

found=0;

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

{

if(j==m[i])

found=1;

}

if(found==0)

printf("%d\t",j);

}

}

OUTPUT:

$ vi bankersdetection.c $ cc bankersdetection.c $ ./a

.out

enter total no of processes: 4

enter clain matrix:

0 1 0 0 1

0 0 1 0 1

0 0 0 0 1

1 0 1 0 1

enter allocation matrix:

1 0 1 1 0

1 1 0 0 0

0 0 0 1 0

0 0 0 0 0

enter resource vector:

2 1 1 2 1

enter availability vector: 0 0 0 0 1

deadlock causing processes are: 0 1

RESULT: Thus the program was executed and verified successfully.

Exercise:12

11 a) AIM: To implement FIFO page replacement technique.

DESCRIPTION: The FIFO page-replacement algorithm is easy to understand and program. However, its¬ performance is not always good. On the one hand, the page replaced may be an initialization module that was useda long¬ time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and¬ is in constant use.

#include<stdio.h>

#include<conio.h>

int fr[3];

void main()

{

void display();

int i,j,page[12]={2,3,2,1,5,2,4,5,3,2,5,2};

int flag1=0,flag2=0,pf=0,frsize=3,top=0;

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

{

fr[i]=-1;

}

for(j=0;j<12;j++)

{

flag1=0; flag2=0; for(i=0;i<12;i++)

{

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

{

flag1=1; flag2=1; break;

}

}

if(flag1==0)

{

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

{

if(fr[i]==-1)

{

fr[i]=page[j]; flag2=1; break;

}

}

}

if(flag2==0)

{

fr[top]=page[j]; top++;

pf++;

if(top>=frsize) top=0;

}

display();

}

printf("Number of page faults : %d ",pf); getch();

}

void display()

{

int i; printf("\n"); for(i=0;i<3;i++)

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

}

**(OR)**

11 a) AIM: To implement FIFO page replacement technique.

**DESCRIPTION**: The FIFO page-replacement algorithm is easy to understand and program. However, its¬ performance is not always good. On the one hand, the page replaced may be an initialization module that was useda long¬ time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and¬ is in constant use.

#include<stdio.h>

#include<conio.h>

main()

{

int i, j, k, f, pf=0, count=0, rs[25], m[10], n;

printf("\n Enter the length of reference string -- ");

scanf("%d",&n);

printf("\n Enter the reference string -- ");

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

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

printf("\n Enter no. of frames -- ");

scanf("%d",&f);

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

m[i]=-1;

printf("\n The Page Replacement Process is -- \n");

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

{

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

{

if(m[k]==rs[i])

break;

}

if(k==f)

{

m[count++]=rs[i];

pf++;

}

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

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

if(k==f)

printf("\tPF No. %d",pf);

printf("\n");

if(count==f)

count=0;

}

printf("\n The number of Page Faults using FIFO are %d",pf);

getch();

}

INPUT

Enter the length of reference string – 20

Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Enter no. of frames -- 3

**OUTPUT**

The Page Replacement Process is

– 7 -1 -1 PF No. 1

7 0 -1 PF No.2

7 0 1 PF No. 3

2 0 1 PF No. 4

2 0 1

2 3 1 PF No. 5

2 3 0 PF No. 6

4 3 0 PF No. 7

4 2 0 PF No. 8

4 2 3 PF No. 9

0 2 3 PF No. 10

0 2 3

0 2 3

0 1 3 PF No. 11

0 1 2 PF No. 12

0 1 2

0 1 2

7 1 2 PF No. 13

7 0 2 PF No. 14

7 0 1 PF No. 15

The number of Page Faults using FIFO are 15

LFU: LEAST FREQUENTLY USED AIM:

11 c)To implement LFU page replacement technique

**ALGORTHIM**: 1. Start Program 2. Read Number Of Pages And Frames 3. Read Each Page Value 4. Search For Page In The Frames 5. If Not Available Allocate Free Frame 6. If No Frames Is Free Repalce The Page With The Page That Is LeastlyUsed 7. Print Page Number Of Page Faults 8. Stop process.

#include<stdio.h>

#include<conio.h>

int fr[3], n, m;

void display();

void main()

{

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

int max,found=0,lg[3],index,k,l,flag1=0,flag2=0,pf=0; float pr;

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

scanf("%d",&n);

printf("Enter the reference string: ");

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

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

printf("Enter no of frames: ");

scanf("%d",&m);

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

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

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

{

flag1=0; flag2=0; for(i=0;i<m;i++)

{

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

{

flag1=1; flag2=1; break;

}

}

if(flag1==0)

{

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

{

if(fr[i]==

-1)

{

fr[i]=page[j]; flag2=1; break; }}}

if(flag2==0) {

for(i=0;i<m;i++) lg[i]=0; for(i=0;i<m;i++) {

for(k=j+1;k<=n;k++) {

if(fr[i]==page[k]) {

lg[i]=k

-j; break;

}}}

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

if(lg[i]==0) {

index=i; found = 1;

break; }}

if(found==0) {

max=lg[0]; index=0; for(i=0;i<m;i++) {

if(max<lg[i]) {

max=lg[i];

index=i; }}}

fr[index]=page[j]; pf++; }

display();

}

printf("Number of page faults : %d\n", pf);

pr=(float)pf/n\*100;

printf("Page fault rate = %f \n", pr);

getch();

}

void display()

{

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

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

printf("\n");

}

OUTPUT: Enter length of the reference string: 12

Enter the reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Enter no of frames: 3

1 -1 -1

1 2 -1

1 2 3

1 2 4

1 2 4

1 2 4

1 2 5

1 2 5

1 2 5

3 2 5

4 2 5

4 2 5

Number of page faults : 7 Page fault rate = 58.333332

11 b) AIM: To implement LRU page replacement technique.

ALGORITHM: 1. Start the process 2. Declare the size 3. Get the number of pages to be inserted 4. Get the value 5. Declare counter and stack 6. Select the least recently used page by counter value 7. Stack them according the selection. 8. Display the values 9. Stop the process

#include<stdio.h>

#include<conio.h>

int fr[3];

void main()

{

void display();

int p[12]={2,3,2,1,5,2,4,5,3,2,5,2},i,j,fs[3];

int index,k,l,flag1=0,flag2=0,pf=0,frsize=3;

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

{

fr[i]=-1;

}

for(j=0;j<12;j++)

{

flag1=0,flag2=0; for(i=0;i<3;i++)

{

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

{

flag1=1; flag2=1; break;

}

}

if(flag1==0)

{

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

{

if(fr[i]==-1)

{

fr[i]=p[j]; flag2=1; break;

}

}

}

if(flag2==0)

{

for(i=0;i<3;i++) fs[i]=0;

for(k=j-1,l=1;l<=frsize-1;l++,k--)

{

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

{

if(fr[i]==p[k]) fs[i]=1;

}

}

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

{

if(fs[i]==0) index=i;

}

fr[index]=p[j]; pf++;

}

display();

}

printf("\n no of page faults :%d",pf);

getch();

}

void display()

{

int i; printf("\n");

for(i=0;i<3;i++) printf("\t%d",fr[i]);

}

**( OR)**

11 b) AIM: To implement LRU page replacement technique.

**DESCRIPTION**

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

#include<stdio.h>

#include<conio.h>

main()

{

int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;

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

scanf("%d",&n);

printf("Enter the reference string -- ");

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

{

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

flag[i]=0;

}

printf("Enter the number of frames -- ");

scanf("%d",&f);

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

{

count[i]=0;

m[i]=-1;

}

printf("\nThe Page Replacement process is -- \n");

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

{

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

{

if(m[j]==rs[i])

{

flag[i]=1;

count[j]=next;

next++;

}

}

if(flag[i]==0)

{

if(i<f)

{

m[i]=rs[i];

count[i]=next;

next++;

}

else

{

min=0;

for(j=1;j<f;j++)

if(count[min] > count[j])

min=j;

m[min]=rs[i];

count[min]=next;

next++;

}

pf++;

}

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

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

if(flag[i]==0)

printf("PF No. -- %d" , pf);

printf("\n");

}

printf("\nThe number of page faults using LRU are %d",pf);

getch();

}

**INPUT**

Enter the length of reference string -- 20

Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Enter the number of frames -- 3

**OUTPUT**

The Page Replacement process is –

7 -1 -1 PF No. -- 1

7 0 -1 PF No. – 2

7 0 1 PF No. – 3

2 0 1 PF No. – 4

2 0 1

2 0 3 PF No. -- 5

2 0 3

4 0 3 PF No. -- 6

4 0 2 PF No. -- 7

4 3 2 PF No. -- 8

0 3 2 PF No. -- 9

0 3 2

0 32

1 3 2 PF No. -- 10

1 3 2

1 0 2 PF No. -- 11

1 0 2

1 0 7 PF No. -- 12

1 0 7

1 0 7

The number of page faults using LRU are 12

11 d) Write a C program to simulate Optimal page replacement algorithm

**DESCRIPTION**

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. The basic idea is to replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

#include<stdio.h>

int n;

main()

{

int

seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s;

int count=1,pf=0,p=0;

float

pfr;

printf("Enter maximum limit of the sequence:");

scanf("%d",&max);

printf("\nEnter the sequence:");

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

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

printf("\nEnter no. of frames:");

scanf("%d",&n);

fr[0]=seq[0];

pf++;

printf("%d\t",fr[0]);

i=1;

while(count<n)

{

flag=1;

p++;

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

{

if(seq[i]==seq[j])

flag=0;

}

if(flag!=0)

{

fr[count]=seq[i];

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

count++;

pf++;

}

i++;

}

printf("\n");

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

{

flag=1;

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

{

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

flag=0;

}

if(flag!=0)

{

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

{

m=fr[j];

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

{

if(seq[k]==m)

{

pos[j]=k;

break;

}

else

pos[j]=1;

}

}

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

{

if(pos[k]==1)

flag=0;

}

if(flag!=0)

s=findmax(pos);

if(flag==0)

{

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

{

if(pos[k]==1)

{

s=k;

break;

}

}

}

fr[s]=seq[i];

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

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

pf++;

printf("\n");

}

}

pfr=(float)pf/(float)max;

printf("\nThe no. of page faults are %d",pf); printf("\nPage fault rate %f",pfr);

getch();

}

int findmax(int a[])

{

int max,i,k=0;

max=a[0];

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

{

if(max<a[i])

{

max=a[i];

k=i;

}

}

return k;

}

INPUT

Enter number of page references – 10

Enter the reference string -- 1 2 3 4 5 2 5 2 5 1 4 3

**Exercise:13**

12a)AIM: Simulate the file allocation strategies using sequential methods

#include<stdio.h>

#include<conio.h>

main()

{

int n,i,j,b[20],sb[20],t[20],x,c[20][20];

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

scanf("%d",&n);

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

{

printf("Enter no. of blocks occupied by file%d",i+1);

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

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

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

t[i]=sb[i];

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

c[i][j]=sb[i]++;

}

printf("Filename\tStart block\tlength\n");

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

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

printf("blocks occupiedare:");

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

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

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

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

printf("\n");

}

getch();

}

OUTPUT:

Enter no.of files: 2

Enter no. of blocks occupied by file1 4

Enter the starting block of file1 2

Enter no. of blocks occupied by file2 10

Enter the starting block of file2 5

Filename Start block length

1 2 4

2 5 1

**12 b) AIM: Simulate the file allocation strategies using file allocation methods(indexed)**

**Indexed allocation description**

In linked allocation it is difficult to maintain FAT – so instead of that method indexed allocation method is used. Indexed allocation method solves all the problems in the linked allocation by bringing all the pointers together into one location called index block.

#include<stdio.h>

#include<conio.h>

main()

{

int n,m[20],i,j,ib[20],b[20][20];

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

scanf("%d",&n);

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

{ printf("Enter index block :",i+1);

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

printf("Enter blocks occupied by file%d:",i+1);

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

printf("enter blocks of file%d:",i+1);

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

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

} printf("\nFile\t index\tlength\n");

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

printf("%d\t%d\t%d\n",i+1,ib[i],m[i]);

printf("blocks occupiedare:");

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

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

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

printf("\t%d--->%d\n",ib[i],b[i][j]);

printf("\n");

}

getch();

}

**OUTPUT**:

Enter no. of files:2

Enter index block 3

Enter blocks occupied by file1: 4

enter blocks of file1:9

4 6 7

Enter index block 5

Enter blocks occupied by file2:2

enter blocks of file2:

10 8

File index length

1 3 4

2 5 2 blocksoccupied are:

file1

3--->9

3--->4

3--->6

3--->7

file2

5--->10

5--->8

12 c) AIM**: Simulate the file allocation strategies using file allocation(linked) methods**

**Linked Allocation description**

Linked allocation of disk space overcomes all the problems of contiguous allocation. In linked allocation each file is a linked list of disk blocks where the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file

#include<stdio.h>

#include<conio.h>

struct file

{

char fname[10];

int start,size,block[10];

}

f[10];

main()

{

int i,j,n;

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

scanf("%d",&n);

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

{

printf("Enter file name:");

scanf("%s",&f[i].fname);

printf("Enter starting block:");

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

f[i].block[0]=f[i].start;

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

scanf("%d",&f[i].size);

printf("Enter block numbers:");

for(j=1;j<=f[i].size;j++)

{

scanf("%d",&f[i].block[j]);

}

}

printf("File\tstart\tsize\tblock\n");

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

{

printf("%s\t%d\t%d\t",f[i].fname,f[i].start,f[i].size);

for(j=0;j<f[i].size;j++)

printf("%d--->",f[i].block[j]);

printf("%d",f[i].block[j]);

printf("\n");

}

getch();

}

**OUTPUT**:

Enter no. of files:2

Enter file name:venkat

Enter starting block:20

Enter no.of blocks:6

Enter block numbers:

4

12

15

45

32

25 .

Enter file name:rajesh

Enter starting block:12

Enter no.of blocks:5

Enter block numbers:

6

5

4

3

2

File start size block

venkat 20 6 20--->4--->12--->15--->45--->32--->25

rajesh 12 5 12--->6--->5--->4--->3--->2

Exercise:14

**/\* Program to simulate single level directory \*/**

**Program 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];

clrscr();

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();

}

**Program Output:**

Enter the directory name:sss

Enter the number of files:3

Enter file name to be created:aaa

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

Enter file name to be created:bbb

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

Enter file name to be created:ccc

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

Directory name is:sss

Files names are:

aaa

bbb

ccc

**Write C programs to simulate the Two level directory File organization technique**#include<stdio.h>

#include<string.h>

#include<stdlib.h>

struct file{

char fileName[15][20];

char dirName[10];

int fno;

};

struct file dir[10];

int i,n;

//FUNCTION TO INSERT A FILE INTO DIRECTORY i

void InsertFile(int i){

printf("n Enter the File name ");

scanf("%s",dir[i].fileName[dir[i].fno]);

dir[i].fno++;

}

//FUNCTION TO DISPLAY ALL FILES AND DIRECTORIES

void DisplayFiles(){

int j;

printf("nnnn");

printf("+------------------------+");

printf("n Directorytfiles | n");

printf("+------------------------+");

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

printf("n %s",dir[j].dirName);

for(i=0;i<dir[j].fno;i++){

printf("n tt%s",dir[j].fileName[i]);

}

printf("n+------------------------+");

}

printf("nnnn");

}

//FUNCTION TO DELETE A FILE FROM DIRECTORY

void DeleteFile(int z){

char name[20];

printf("n Enter the file to be deleted : ");

scanf("%s",name);

for(i=0;i<dir[z].fno;i++){

if(strcmp(dir[z].fileName[i],name)==0){

printf("%s is deleted t",dir[z].fileName[i]);

strcpy(dir[z].fileName[i],dir[z].fileName[dir[z].fno-1]);

dir[z].fno--;

}

}

}

//FUNCTION TO SEARCH FOR A FILE

void SearchFile(int z){

char name[20];

int found=-1;

printf("n Enter the file to be searched :");

scanf("%s",name);

for(i=0;i<dir[z].fno;i++){

if(strcmp(dir[z].fileName[i],name)==0){

printf("n The File is found at position %dn",i+1);

found=1;

break;

}

}

if(found==-1)

printf("n The file is not found ");

}

//FUNCTION TO FIND THE INDEX OF A DIRECTORY FOR A GIVEN name

int getdir(char name[30]){

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

if(strcmp(dir[i].dirName,name)==0){

return i;

break;

}

}

return i;

}

//FUNCTION TO CHOOSE AN OPERATION

int option(int i){

while(1){

int op;

printf("nchoose the option n1:Insert a file in directory %s",dir[i].dirName);

printf("n2:Display Files and Directories");

printf("n3:Delete File from %s",dir[i].dirName);

printf("n4:Search File in %s",dir[i].dirName);

printf("n5:choose another directoryn6:Exitn>>> ");

scanf("%d",&op);

switch(op){

case 1:InsertFile(i);

break;

case 2:DisplayFiles();

break;

case 3:DeleteFile(i);

break;

case 4:SearchFile(i);

break;

case 5:return 1;

break;

case 6:exit(0);

}

}

}

int main(){

char name[20];

printf("n Enter the no of directories ");

scanf("%d",&n);

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

dir[i].fno=0;

printf("nFor directory %d Enter the directory name: ",i+1);

scanf("%s",dir[i].dirName);

}

while(1){

printf("n choose a directory to perform operation ");

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

printf("nt%s",dir[i].dirName);

}

printf("n>>");

scanf("%s",name);

option(getdir(name));

}

return 0;

}

#### Output

Enter the no of directories 2

For directory 1 Enter the directory name: Programs

For directory 2 Enter the directory name: Notes

choose a directory to perform operation

Programs

Notes

>>Programs

choose the option

1:Insert a file in directory Programs

2:Display Files and Directories

3:Delete File from Programs

4:Search File in Programs

5:choose another directory

6:Exit

>>> 1

**Write C programs to simulate the Hierarchical  directory File organization technique**

#include<stdio.h>  
#include<graphics.h>  
#include<string.h>  
struct tree\_element  
{  
char name[20];  
int x, y, ftype, lx, rx, nc, level;  
struct tree\_element \*link[5];  
};  
typedef struct tree\_element node;  
void main()  
{  
int gd=DETECT,gm;  
node \*root;  
root=NULL;  
create(&root,0,"root",0,639,320);  
clrscr();  
initgraph(&gd,&gm,"c:\tc\BGI");  
display(root);  
closegraph();  
}  
create(node \*\*root,int lev,char \*dname,int lx,int rx,int x)  
{  
int i, gap;  
if(\*root==NULL)  
{  
(\*root)=(node \*)malloc(sizeof(node));  
printf("Enter name of dir/file(under %s) : ",dname);  
fflush(stdin);  
gets((\*root)->name);  
printf("enter 1 for Dir/2 for file :");  
scanf("%d",&(\*root)->ftype);  
(\*root)->level=lev;  
(\*root)->y=50+lev\*50;  
(\*root)->x=x;  
(\*root)->lx=lx;  
(\*root)->rx=rx;  
for(i=0;i<5;i++)  
(\*root)->link[i]=NULL;  
if((\*root)->ftype==1)  
{  
printf("No of sub directories/files(for %s):",(\*root)->name); scanf("%d",&(\*root)>nc);  
if((\*root)->nc==0)  
gap=rx-lx;  
else  
gap=(rx-lx)/(\*root)->nc;  
for(i=0;i<(\*root)->nc;i++)  
create(&((\*root)>link[i]),lev+1,(\*root)>name,lx+gap\*i,lx+gap\*i+gap,  
lx+gap\*i+gap/2);  
}  
else  
(\*root)->nc=0;  
}  
}  
display(node \*root)  
{  
int i;  
settextstyle(2,0,4);  
settextjustify(1,1);  
setfillstyle(1,BLUE);  
setcolor(14);  
if(root !=NULL)  
{  
for(i=0;i<root->nc;i++)  
line(root->x,root->y,root->link[i]->x,root->link[i]->y);  
if(root->ftype==1)  
bar3d(root->x-20,root->y-10,root->x+20,root>y+10,0,0);  
else  
fillellipse(root->x,root->y,20,20);  
outtextxy(root->x,root->y,root->name);  
for(i=0;i<root->nc;i++)  
display(root->link[i]);  
}  
}