**INDEX :-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EXP.NO** | **DATE:** | **NAME OF THE EXPERIMENT** | **PAGENO:** | **REMARKS:** |
| 1.) |  | Simulate the following CPU scheduling algorithms  a) Round Robin  b) SJF  c) FCFS  d) Priority | 2 - 23 |  |
| 2.) |  | Multiprogramming-Memory management- Implementation of fork (), wait (), exec() and  exit (), System calls | 24 - 30 |  |
| 3.) |  | Simulate the following  a) Multiprogramming with a fixed number of tasks (MFT)  b) Multiprogramming with a variable number of tasks (MVT) | 31 – 37 |  |
| 4.) |  | Simulate Bankers Algorithm for Dead Lock Avoidance | 38 – 42 |  |
| 5.) |  | Simulate Bankers Algorithm for Dead Lock Prevention. | 43 - 50 |  |
| 6.) |  | Simulate the following page replacement algorithms.  a) FIFO b) LRU c) LFU | 51 - 60 |  |
| 7.) |  | Simulate the following File allocation strategies a) Sequenced b) Indexed c) Linked | 61 - 71 |  |

EXERCISE-1: **Simulate the following CPU scheduling algorithms:**

**a) FCFS b) SJF c) Round Robin d) Priority**

1a.)  **FCFS:**

**Aim**: To write a program for implementing FCFS Scheduling algorithm.

**Definition**: FCFS is an operating system scheduling algorithm that automatically

executes queued request and processes in order of their arrival time (A.T).

**Algorithm**:

1.) Start the process

2.) Declare the array size

3.) Get the number of elements to be inserted

4.) Select the process that arrived first in the ready queue

5.) Make the average waiting the length of the next process

6.) Start with the first process from its selection as above and let other process to be in queue

7.) Calculate the total number of burst time

8.) Display the values

9.) Stop the process

**Program**:

#include<stdio.h>

main()

{

float avgwt,avgtt;

char pname[10][10],c[10][10];

int wt[10],tt[10],bt[10],at[10],t,q,i,n,sum=0,sbt=0,ttime,j,ss=0;

printf("\n\n Enter the number of processes: ");

scanf("%d",&n);

printf("\n\n Enter the NAME , BURST TIME and ARRIVAL TIME of the process");

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

{

printf("\n\n NAME : ");

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

printf("\n\n BURST TIME : ");

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

printf("\n\n ARRIVAL TIME : ");

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

}

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

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

{

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

{

t=at[i];

at[i]=at[j];

at[j]=t;

q=bt[i];

bt[i]=bt[j];

bt[j]=q;

strcpy(c[i],pname[i]);

strcpy(pname[i],pname[j]);

strcpy(pname[j],c[i]);

}

}

wt[0]=0;

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

{

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

sum=sum+(wt[i]-at[i]);

sbt=sbt+(wt[i+1]-at[i]);

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

ss=ss+bt[i];

}

avgwt=(float) sum/n;

avgtt=(float)sbt/n;

printf("\n\n Average waiting time = %f",avgwt);

printf("\n\n Average turn-around time = %f",avgtt);

printf("\n\n GANTT CHART\n");

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

printf("|\t%s\t",pname[i]);

printf("\n");

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

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

printf("%d\n",ss);

printf("\n");

}

**OUTPUT:**

[root@localhost ~]# ./a.out

Enter the number of processes: 4

Enter the NAME , BURST TIME and ARRIVAL TIME of the process

NAME : p1  
 BURST TIME : 4  
 ARRIVAL TIME : 0  
  
 NAME : p2  
 BURST TIME : 9  
 ARRIVAL TIME : 2  
  
 NAME : p3  
 BURST TIME : 8  
 ARRIVAL TIME : 4

NAME : p4  
 BURST TIME : 3  
 ARRIVAL TIME : 3  
 Average waiting time = 6.000000  
 Average turn-around time = 12.000000  
 **GANTT CHART**:

| p1 | p2 | p4 | p3  
0 4 13 16 24

**1b.) SJF:**

**Aim**: To write a program for implementing SJF Scheduling algorithm.

**Definition**: SJF is an operating system scheduling algorithm, which works on the

Process with shortest time (or) duration first.

**Algorithm**:

1.) Start the process

2.) Declare the array size

3.) Get the number of elements to be inserted

4.) Select the process which has shortest burst will execute first

5.) If the two process have the same burst time then FCFS scheduling is used

6.) Make the average waiting the length of the next process

7.) Start with the first process from its selection as above and let other process to be in queue

8.) Calculate the total number of burst time

9.) Display the values

10.) Stop the process

**Program**:

#include<stdio.h>

main()

{

float avgwt,avgtt;

char pname[10][10],c[10][10];

int wt[10],tt[10],bt[10],at[10],t,q,i,n,sum=0,sbt=0,ttime,j,ss=0;

printf("\n\n Enter the number of processes: ");

scanf("%d",&n);

printf("\n\n Enter the NAME, BURSTTIME, and ARRIVALTIME of the processes ");

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

{

printf("\n\n NAME : ");

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

printf("\n\n BURST TIME : ");

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

printf("\n\n ARRIVAL TIME : ");

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

}

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

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

{

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

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

{

t=at[i];

at[i]=at[j];

at[j]=t;

q=bt[i];

bt[i]=bt[j];

bt[j]=q;

strcpy(c[i],pname[i]);

strcpy(pname[i],pname[j]);

strcpy(pname[j],c[i]);

}

if(at[i]!=at[j])

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

{

t=at[i];

at[i]=at[j];

at[j]=t;

q=bt[i];

bt[i]=bt[j];

bt[j]=q;

strcpy(c[i],pname[i]);

strcpy(pname[i],pname[j]);

strcpy(pname[j],c[i]);

}

}

wt[0]=0;

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

{

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

sum=sum+(wt[i]-at[i]);

sbt=sbt+(wt[i+1]-at[i]);

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

ss=ss+bt[i];

}

printf("\n\n GANTT CHART");

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

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

printf("|\t%s\t",pname[i]);

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

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

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

printf("%d\n",ss);

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

printf("\n\n Total WAITING TIME = %d ",sum);

printf("\n\n Total TURNAROUND TIME = %d ",sbt);

avgwt=(float)sum/n;

avgtt=(float)sbt/n;

printf("\n\n Average WAITING TIME = %f ",avgwt);

printf("\n\n Average TURNAROUND TIME = %f ",avgtt);

}

**OUTPUT:**

Enter the number of processes: 5  
Enter the NAME, BURSTTIME, and ARRIVALTIME of the processes

NAME : p0  
 BURST TIME : 2  
 ARRIVAL TIME : 0

NAME : p1  
 BURST TIME : 4  
 ARRIVAL TIME : 0

NAME : p2  
 BURST TIME : 5  
 ARRIVAL TIME : 0

NAME : p3  
BURST TIME : 6  
ARRIVAL TIME : 0

NAME : p4  
BURST TIME : 8  
ARRIVAL TIME : 0

GANTT CHART

----------------------------------------------------------------------   
| p0 | p1 | p2 | p3 | p4  
-----------------------------------------------------------------------  
0 2 6 11 17 25

Total WAITING TIME = 36

Total TURNAROUND TIME = 61   
Average WAITING TIME = 7.200000

1c.) **Round Robin**

**Aim**: To write a program for implementing Round robin Scheduling algorithm.

**Definition**: Round robin is an operating system scheduling algorithm that

automatically executes queued request and processes in order of their

arrival time (A.T).

**Algorithm**:

1.) Start the process

2.) Declare the array size

3.) Get the number of elements to be inserted

4.) Get the value

5.) Set the time sharing system with preemption

6.) Define quantum is defined from 10 to 100

7.) Declare the queue as a circular

8.) Make the CPU scheduler goes around the ready queue allocating CPU to each process for the time interval specified.

9.) Make the CPU scheduler picks first process and sets time to interrupt after quantum expired dispatches the process.

10.) If the process has burst less than the time quantum than the process releases the

CPU.

**Program**:

#include<stdio.h>

main()

{

int pt[10][10],a[10][10],at[10],pname[10][10],i,j,n,k=0,q,sum=0;

float avg;

printf("\n\n Enter the number of processes : ");

scanf("%d",&n);

for(i=0;i<10;i++)

{

for(j=0;j<10;j++)

{

pt[i][j]=0;

a[i][j]=0;

}

}

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

{

j=0;

printf("\n\n Enter the process time for process %d : ",i+1);

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

}

printf("\n\n Enter the time slice : ");

scanf("%d",&q);

printf("\n\n");

for(j=0;j<10;j++)

{

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

{

a[2\*j][i]=k;

if((pt[i][j]<=q)&&(pt[i][j]!=0))

{

pt[i][j+1]=0;

printf(" %d P%d %d\n",k,i+1,k+pt[i][j]);

k+=pt[i][j];

a[2\*j+1][i]=k;

}

else if(pt[i][j]!=0)

{

pt[i][j+1]=pt[i][j]-q;

printf(" %d P%d %d\n",k,i+1,(k+q));

k+=q;

a[2\*j+1][i]=k;

}

else

{

a[2\*j][i]=0;

a[2\*j+1][i]=0;

}

}

}

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

sum+=a[0][i];

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

{

for(j=1;j<10;j++)

{

if((a[j][i]!=0)&&(a[j+1][i]!=0)&&((j+1)%2==0))

sum+=((a[j+1][i]-a[j][i]));

}

}

avg=(float)sum/n;

printf("\n\n Average waiting time = %f msec",avg);

sum=avg=0;

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

{

i=1;

while(a[i][j]!=0)

i+=1;

sum+=a[i-1][j];

}

avg=(float)sum/n;

printf("\n\n Average turnaround time = %f msec\n\n",avg);

}

**OUTPUT:**

[root@localhost ~]# ./a.out

Enter the number of processes : 4

Enter the process time for process 1 : 8  
 Enter the process time for process 2 : 3  
 Enter the process time for process 3 : 6  
 Enter the process time for process 4 : 1

Enter the time slice : 2

0 P1 2  
 2 P2 4  
 4 P3 6  
 6 P4 7  
 7 P1 9  
 9 P2 10  
 10 P3 12  
 12 P1 14  
 14 P3 16  
 16 P1 18

Average waiting time = 8.250000 msec  
 Average turnaround time = 12.750000 msec

**1d.) Priority:**

**Aim**: To write a program for implementing priority Scheduling algorithm.

**Definition**: Priority scheduling is non preemptive algorithm and one of the most

common scheduling algorithm in batch.

**Algorithm**:

1.) Start the process

2.) Declare the array size

3.) Get the number of elements to be inserted

4.) Get the priority for each and value of process

4.) Start the higher priority process from its initial position let other process to be queue.

5.) Calculate the total number of burst time

6.) Display the values

7.) Stop the process

**Program**:

#include<stdio.h>

main()

{

float avgwt,avgtt;

char pname[10][10],c[10][10];

int wt[10],tt[10],bt[10],pt[10],t,q,i,n,sum=0,sbt=0,ttime,j,ss=10;

printf("\n\n Enter the number of processes : ");

scanf("%d",&n);

printf("\n\n Enter the NAME and BURSTTIME ");

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

{

printf("\n\n NAME : ");

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

printf("\n\n BURSTTIME : ");

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

}

printf("\n\n Enter the priorities of the processes ");

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

{

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

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

}

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

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

{

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

{

t=pt[i];

pt[i]=pt[j];

pt[j]=t;

q=bt[i];

bt[i]=bt[j];

bt[j]=q;

strcpy(c[i],pname[i]);

strcpy(pname[i],pname[j]);

strcpy(pname[j],c[i]);

}

}

wt[0]=0;

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

{

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

sum=sum+wt[i];

sbt=sbt+wt[i+1];

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

ss=ss+bt[i];

}

printf("\n\n GANTT CHART");

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

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

printf("|\t%s\t",pname[i]);

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

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

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

printf("%d\n",ss);

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

printf("\n\n Total WAITING TIME of the process = %d",sum);

printf("\n\n Total TURNAROUND TIME of the process = %d",sbt);

avgwt=(float)sum/n;

avgtt=(float)sbt/n;

printf("\n\n Average WAITING TIME of the process = %f",avgwt);

printf("\n\n Average TURNAROUND TIME of the process = %f",avgtt);

}

**OUTPUT:**

[root@localhost ~]# ./a.out  
 Enter the number of processes : 4  
 Enter the NAME and BURSTTIME

NAME : p1  
 BURSTTIME : 8

NAME : p2  
 BURSTTIME : 3

NAME : p3  
 BURSTTIME : 6

NAME : p4  
 BURSTTIME : 1

Enter the priorities of the processes   
 Priority of process1 : 1  
 Priority of process2 : 5  
 Priority of process3 : 2  
 Priority of process4 : 4  
  
 GANTT CHART  
------------------------------------------------------

| p1 | p3 | p4 | p2  
------------------------------------------------------

0 8 14 15 28

Total WAITING TIME of the process = 37

Total TURNAROUND TIME of the process = 55  
 Average WAITING TIME of the process = 9.250000  
 Average TURNAROUND TIME of the process = 13.750000

*EXERCISE-2*: **Multiprogramming-Memory management- Implementation of fork (), wait (), exec() and exit (), System calls**.

**Aim**: To implement -Memory management-Implementation of fork(),wait(),exec()&exit(),system calls.

**Definition**: Memory management: It is defined as all data in memory before and after processing , it is used for optimizing CPU utilization & computer response to users.

Systemcalls: System calls provides services of the OS to the user programs via application interface(API). It provides an interface b/w a process & OS.

**Implementation of fork() system call using C program:**

Fork(): It is used to create a new process, which consist of a copy of the address space of the parent.

**Algorithm**:

1.) Start

2.) Declare the process

3.) Call the fork function for the process. If A process equal to zero print child.

4.) Else if print parent process if process id greater than zero

5.) else print error & return 0,if process id less than zero

6.) Stop

**Program**:

#include<sys/types.h>

main()

{

pid\_t pid;

pid = fork();

if(pid==0)

printf(“\n I am the child process:”);

else if (pid>0)

printf(“\n I am the parent process My child pid is %d”,pid);

else

perror(“error in fork”);

}

**Output:** I am the parent process My child pid is 7005

**Implementation of wait() system call using C program:**

Wait(): It is the method to wait for the children to terminate.

**Algorithm:**

1. Start
2. Declare the process
3. Call the fork() function for the process. If process equal to ‘-1’ then exit process
4. If process equal to zero print child class
5. Call wait() method to terminate child process
6. Then print parent class
7. Stop

**Program:**

#include <stdio.h>

void main()

{

int pid, status;

pid= fork();

if (pid == -1)

{

printf (" fork failed \n"),

Exit(1);

}

if (pid==0)

{

/\* Child \*/ printf ("child here! \n" );

}

else

{

/\*parent \*/ wait (&Status);

printf (" well done child! \n");

}

}

**Output:**

child here

well done child.

**Implementation of Exec() system call using C program:**

Exec(): overwrites the process address Space with a new program

**Algorithm:**

1.) Start

2.) Declare process

3.) print parent process & child process

4.) Terminate child process

5.) Call exec() System call & Terminate the parent process

6.) Replace The above process with ExecVp()

7.) Stop

**Program:**

#include <stdio.h>

#include < sys/wait.h>

# include <unistd.h>

# include<stdlib.h>

int (main)

{

if (fork( )==0)

{

print (“ HC: hello from child \n”);

Exit (1);

}

Else

{

printf (“ HP : hello from parent \n");

wait (Null);

printf (“CT: child has terminated \n”);

}

//exec() System call

char \* args[] = { "./Exec" Null }; // calling of exec process

execvp (args [o], args);

printf (“ Terminating parent process Bye \n”);

return 0;

}

// execvp System call replaces. This process

# include<stdio.h>

# include<unistd.h>

int main()

{

printf (“ \n Welcome to exec.c process");

printf (" \n I am EXEC.C Called by execvp()”);

Return 0;

}

Exercise-3**: Simulate the following**

**a) Multiprogramming with a fixed number of tasks (MFT)**

**b) Multiprogramming with a variable number of tasks (MVT)**

**3a.)** **Aim**: To write a program on multiprogramming with a fixed number of tasks (MFT).

**Description**: Multiprogramming with a fixed number of tasks is one of the old memory management techniques in which the memory partitioned into fixed size partition &each job is assigned to a partition does not change.

**Algorithm**:

1.) Start the process

2.) Declare a variables

3.) Enter the total memory size (in bytes)

4.) Allocate memory for OS

5.) Read the number of partitions to be divided ‘p’ partition size i.e.,s=m/p

6.) Read the number partition & process size

7.) If process size is less than partition size allot else block the process while allocating update memory wastage-external fragmentation.

8.) Print the result

**Program**:

#include<stdio.h>

#include<conio.h>

int main()

{

int m,p,s,p1;

int m1[4],i,f,f1=0,f2=0,fra1,fra2,s1,pos;

clrscr();

printf("Enter the memory size:");

scanf("%d",&m);

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

scanf("%d",&p);

s=m/p;

printf("Each partn size is:%d",s);

printf("\nEnter the no of processes:");

scanf("%d",&p1);

pos=m;

for(i=0;i<p1;i++)

{

if(pos<s)

{

printf("\nThere is no further memory for process%d",i+1);

m1[i]=0;

break;

}

else

{

printf("\nEnter the memory req for process%d:",i+1);

scanf("%d",&m1[i]);

if(m1[i]<=s)

{

printf("\nProcess is allocated in partition%d",i+1);

fra1=s-m1[i];

printf("\nInternal fragmentation for process is:%d",fra1);

f1=f1+fra1;

pos=pos-s;

}

else

{

printf("\nProcess not allocated in partition%d",i+1);

s1=m1[i];

while(s1>s)

{

s1=s1-s;

pos=pos-s;

}

pos=pos-s;

fra2=s-s1;

f2=f2+fra2;

printf("\nExternal Fragmentation for this process is:%d",fra2);

}

}

}

printf("\nProcess\tallocatedmemory");

for(i=0;i<p1;i++)

printf("\n%5d\t%5d",i+1,m1[i]);

f=f1+f2;

printf("\nThe tot no of fragmentation is:%d",f);

getch();

return 0;

}

**OUTPUT:**

Enter the memory size: 80

Enter the no of partitions: 4

Each partition size: 20

Enter the number of processes: 2

Enter the memory req for process1: 18

Process1 is allocated in partn1

Internal fragmentation for process1 is: 2

Enter the memory req for process2: 22

Process2 is not allocated in partn2

External fragmentation for process2 is: 18

**3b.) Multiprogramming with a variable number of tasks (MVT):**

**Aim**: To implement a program on multiprogramming with variable number of tasks (MVT).

**Description**: Multiprogramming with a Variable number of tasks is the memory management technique of the old memory management technique in which each job gets just the amount of memory it needs. That is partitioning of memory is dynamic & changes as jobs enter & leave the system. MVT is a more “efficient” user of resources.

**Algorithm**:

1.) Start the process

2.) Declare a variables

3.) Enter the total memory size

4.) Allocate memory for OS

5.) Read the number of partitions to be divided ‘n’ partition

6.) Read the process number and & process size

7.) If process size is less than partition size allot else block the process while allocating update memory wastage-external fragmentation.

8.) Print the result

9.) Stop the process

**Program**:

#include<stdio.h>

#include<conio.h>

main()

{

int ms,mp[10],i, temp,n=0;

char ch = 'y';

clrscr();

printf("\nEnter the total memory available (in Bytes)-- ");

scanf("%d",&ms);

temp=ms;

for(i=0;ch=='y';i++,n++)

{

printf("\nEnter memory required for process %d (in Bytes) -- ",i+1);

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

if(mp[i]<=temp)

{

printf("\nMemory is allocated for Process %d ",i+1);

temp = temp - mp[i];

}

else

{

printf("\nMemory is Full");

break;

}

printf("\nDo you want to continue(y/n) -- ");

scanf(" %c", &ch);

}

printf("\n\nTotal Memory Available -- %d", ms);

printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED ");

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

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

printf("\n\nTotal Memory Allocated is %d",ms-temp);

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

getch( );

}

INPUT:

Enter the total memory available (in Bytes) -- 1000

Enter memory required for process 1 (in Bytes) -- 400

Memory is allocated for Process 1

Do you want to continue(y/n) -- y

Enter memory required for process 2 (in Bytes) -- 275

Memory is allocated for Process 2

Do you want to continue(y/n) -- y

Enter memory required for process 3 (in Bytes) -- 550

**OUTPUT:**

Memory is Full

Total Memory Available -- 1000

PROCESS MEMORY-ALLOCATED

1 400

2 275

Total Memory Allocated is 675

Total External Fragmentation is 325

**Exercise-4: Simulate Bankers Algorithm for Dead Lock Avoidance**

**Aim**: To simulate Bankers algorithm for Deadlock avoidance.

**Description**: The Bankers algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amount of all resources, then makes an ‘S-State’ check to test for possible activities, before deciding whether allocation should be allowed to continue.

**Algorithm**:

1.) Start the program

2.) Get the values of resources & processes

3.) Get the available value

4.) After allocation find the need value

5.) Check whether its possible to allocate

6.) If its possible then the system is in safe state

7.) Else system is not in safety state

8.) If the new request comes then check that the system is in safety.

9.) or not if we allow the request

10.) Stop the program

11.) End

* Before going to built a program on Bankers algorithm for deadlock avoidance, we must know the safety algorithm steps.

**Safety algorithim:**

1. Initially work=Available

finish[i] = false, where i= 1,2,3,4

i.e., The process will not Executed.

2.) Find an ‘i’ Such that

a) finish[ i ] = false then find

b) needs<= work

1. if need <=work, then
2. work = work + allocation

finish [ i ] = true,

i.e., it is executed, then repeat the above step.

1. if finish [ i ] == true for all ‘i’, then System is in safe state i.e., it avoid the deadlocks in process while Executing.

**Program**:

#include<stdio.h>

#include<conio.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");

    }

    getch();

}

**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:2

Enter the requested resource:1 1 2

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

1 2

process 4:

1 0

process 5:

1 0

Deadlock will occur

**Experiment-5 Q) simulate. Bankers algorithm for Deadlock prevention.**

**Aim:** To simulate Bankers algorithm -for Deadlock prevention.

**Definition:** When a process enters into a system, it must declare maximum number of instance of each resource type it needed. This number may exceeds the total number of resources in system. When the user request a set of resources, The system must determine whether the allocation of each resources will leave the System in safe state.

**Algorithm:**

1.) Start the program (or) process

2.) Get The value of resources and processes

3.) Get The available value

4.)After allocation find the need value

5.) check whether it is 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 the process

**Program:**

#include< stdio.h>

#include< conio.h>

void main()

{

int allocated[15][15],max[15][15],need[15][15],avail[15],tres[15],work[15],flag[15];

int pno,rno,i,j,prc,count,t,total;

count=0;

clrscr();

printf("\n Enter number of process:");

scanf("%d",&pno);

printf("\n Enter number of resources:");

scanf("%d",&rno);

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

{

flag[i]=0;

}

printf("\n Enter total numbers of each resources:");

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

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

printf("\n Enter Max resources for each process:");

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

{

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

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

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

}

printf("\n Enter allocated resources for each process:");

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

{

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

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

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

}

printf("\n available resources:\n");

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

{

avail[j]=0;

total=0;

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

{

total+=allocated[i][j];

}

avail[j]=tres[j]-total;

work[j]=avail[j];

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

}

do

{

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

{

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

{

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

}

}

printf("\n Allocated matrix Max need");

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

{

printf("\n");

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

{

printf("%4d",allocated[i][j]);

}

printf("|");

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

{

printf("%4d",max[i][j]);

}

printf("|");

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

{

printf("%4d",need[i][j]);

}

}

prc=0;

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

{

if(flag[i]==0)

{

prc=i;

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

{

if(work[j]< need[i][j])

{

prc=0;

break;

}

}

}

if(prc!=0)

break;

}

if(prc!=0)

{

printf("\n Process %d completed",i);

count++;

printf("\n Available matrix:");

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

{

work[j]+=allocated[prc][j];

allocated[prc][j]=0;

max[prc][j]=0;

flag[prc]=1;

printf(" %d",work[j]);

}

}

}while(count!=pno&&prc!=0);

if(count==pno)

printf("\nThe system is in a safe state!!");

else

printf("\nThe system is in an unsafe state!!");

getch();

}

**OUTPUT:**

Enter number of process: 5

Enter number of resources: 3

Enter total numbers of each resources: 10 5 7

Enter Max resources for each process:

for process 1: 7 5 3

for process 2: 3 2 2

for process 3: 9 0 2

for process 4: 2 2 2

for process 5: 4 3 3

Enter allocated resources for each process:

for process 1: 0 1 0

for process 2: 3 0 2

for process 3: 3 0 2

for process 4: 2 1 1

for process 5: 0 0 2

available resources:

2 3 0

Allocated matrix Max need

0 1 0| 7 5 3| 7 4 3

3 0 2| 3 2 2| 0 2 0

3 0 2| 9 0 2| 6 0 0

2 1 1| 2 2 2| 0 1 1

0 0 2| 4 3 3| 4 3 1

Process 2 completed

Available matrix: 5 3 2

Allocated matrix Max need

0 1 0| 7 5 3| 7 4 3

0 0 0| 0 0 0| 0 0 0

3 0 2| 9 0 2| 6 0 0

2 1 1| 2 2 2| 0 1 1

0 0 2| 4 3 3| 4 3 1

Process 4 completed

Available matrix: 7 4 3

Allocated matrix Max need

0 1 0| 7 5 3| 7 4 3

0 0 0| 0 0 0| 0 0 0

3 0 2| 9 0 2| 6 0 0

0 0 0| 0 0 0| 0 0 0

0 0 2| 4 3 3| 4 3 1

Process 1 completed

Available matrix: 7 5 3

Allocated matrix Max need

0 0 0| 0 0 0| 0 0 0

0 0 0| 0 0 0| 0 0 0

3 0 2| 9 0 2| 6 0 0

0 0 0| 0 0 0| 0 0 0

0 0 2| 4 3 3| 4 3 1

Process 3 completed

Available matrix: 10 5 5

Allocated matrix Max need

0 0 0| 0 0 0| 0 0 0

0 0 0| 0 0 0| 0 0 0

0 0 0| 0 0 0| 0 0 0

0 0 0| 0 0 0| 0 0 0

0 0 2| 4 3 3| 4 3 1

Process 5 completed Available matrix: 10 5 7 The system is in a safe state!!

**Exercise-6a: IMPLEMENTATION OF FIFO PAGE REPLACEMENT**

**ALGORITHM**

**AIM:**

To write a c program to implement FIFO page replacement algorithm

**ALGORITHM:**

1. Start the process

2. Declare the size with respect to page length

3. Check the need of replacement from the page to memory

4. Check the need of replacement from old page to new page in memory

5. Forma queue to hold all pages

6. Insert the page require memory into the queue

7. Check for bad replacement and page fault

8. Get the number of processes to be inserted

9. Display the values

10. Stop the process

**PROGRAM:**

#include<stdio.h>

int main()

{

int i,j,n,a[50],frame[10],no,k,avail,count=0;

printf("\n ENTER THE NUMBER OF PAGES:\n");

scanf("%d",&n);

printf("\n ENTER THE PAGE NUMBER :\n");

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

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

printf("\n ENTER THE NUMBER OF FRAMES :");

scanf("%d",&no);

for(i=0;i<no;i++)

frame[i]= -1;

j=0;

printf("\tref string\t page frames\n");

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

{

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

avail=0;

for(k=0;k<no;k++)

if(frame[k]==a[i])

avail=1;

if (avail==0)

{

frame[j]=a[i];

j=(j+1)%no;

count++;

for(k=0;k<no;k++)

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

}

printf("\n");

}

printf("Page Fault Is %d",count);

return 0;

}

**OUTPUT:**

ENTER THE NUMBER OF PAGES:  20

ENTER THE PAGE NUMBER :       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

ref string        page frames

7                7      -1      -1

0                7       0      -1

1                7       0       1

2                2       0       1

0

3                2       3       1

0                2       3       0

4                4       3       0

2                4       2       0

3                 4       2       3

0                0       2       3

3

2

1                 0       1       3

2                 0       1       2

0

1

7                 7       1       2

0                7       0       2

1                7       0       1

Page Fault Is 15

**Exercise-6b.):- IMPLEMENTATION OF LRU PAGE REPLACEMENT ALGORITHM AND PROGRAM**

**Aim:** To simulate LRU page replacement algorithm.

**Description:** This page algorithm replaces the page which has not been referred to a long time. This just opposite to optimal page replacement algorithm.

**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

**PROGRAM:**

#include<stdio.h>

main()

{

int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];

printf("Enter no of pages:");

scanf("%d",&n);

printf("Enter the reference string:");

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

            scanf("%d",&p[i]);

printf("Enter no of frames:");

scanf("%d",&f);

q[k]=p[k];

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

c++;

k++;

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

            {

                        c1=0;

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

                        {

                                    if(p[i]!=q[j])

                                    c1++;

                        }

                        if(c1==f)

                        {

                                    c++;

                                    if(k<f)

                                    {

                                                q[k]=p[i];

                                                k++;

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

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

                                                printf("\n");

                                    }

                                    else

                                    {

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

                                                {

                                                            c2[r]=0;

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

                                                            {

                                                            if(q[r]!=p[j])

                                                            c2[r]++;

                                                            else

                                                            break;

                                                }

                                    }

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

                                     b[r]=c2[r];

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

                                    {

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

                                                {

                                                            if(b[r]<b[j])

                                                            {

                                                                        t=b[r];

                                                                        b[r]=b[j];

                                                                        b[j]=t;

                                                            }

                                                }

                                    }

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

                                    {

                                                if(c2[r]==b[0])

                                                q[r]=p[i];

                                                printf("\t%d",q[r]);

                                    }

                                    printf("\n");

                        }

            }

}

printf("\nThe no of page faults is %d",c);

}

**OUTPUT:**

Enter no of pages:10

Enter the reference string:7 5 9 4 3 7 9 6 2 1

Enter no of frames:3

        7

        7       5

        7       5       9

        4       5       9

        4       3       9

        4       3       7

        9       3       7

        9       6       7

        9       6       2

        1       6       2

The no of page faults is 10

**6c.) Simulate the LFU replacement algorithm.**

**Aim:** To simulate LFU page replacement algorithm.

**Description:** **Least Frequently Used (LFU)** is a caching algorithm in which the least frequently used cache block is removed whenever the cache is overflowed. In LFU we check the old page as well as the frequency of that page and if the frequency of the page is larger than the old page we cannot remove it and if all the old pages are having same frequency then take last i.e., FIFO method for that and remove that page.

**Algorithm:**

1.) Start the process

2.) Declare the size

3.) Get the number of pages to be inserted

4.) Get the value

5.) The simplest method to employ an LFU algorithm is to assign a counter to every block that is loaded into the cache.

6.)  Each time a reference is made to that block the counter is increased by one

7.) When the cache reaches capacity and has a new block waiting to be inserted the system will search for the block with the lowest counter and remove it from the cache.

**Ideal LFU**: there is a counter for each item in the catalogue.

**Practical LFU**: there is a counter for the items stored in cache. The counter is forgotten if the item is evicted.

8.) Display the values

9.) Stop the process

**Program:**

#include<stdio.h>

int main()

{

      int total\_frames, total\_pages, hit = 0;

      int pages[25], frame[10], arr[25], time[25];

      int m, n, page, flag, k, minimum\_time, temp;

      printf("Enter Total Number of Pages:\t");

      scanf("%d", &total\_pages);

      printf("Enter Total Number of Frames:\t");

      scanf("%d", &total\_frames);

      for(m = 0; m < total\_frames; m++)

      {

            frame[m] = -1;

      }

      for(m = 0; m < 25; m++)

      {

            arr[m] = 0;

      }

      printf("Enter Values of Reference String\n");

      for(m = 0; m < total\_pages; m++)

      {

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

            scanf("%d", &pages[m]);

      }

      printf("\n");

      for(m = 0; m < total\_pages; m++)

      {

            arr[pages[m]]++;

            time[pages[m]] = m;

            flag = 1;

            k = frame[0];

            for(n = 0; n < total\_frames; n++)

            {

                  if(frame[n] == -1 || frame[n] == pages[m])

                  {

                        if(frame[n] != -1)

                        {

                              hit++;

                        }

                        flag = 0;

                        frame[n] = pages[m];

                        break;

                  }

                  if(arr[k] > arr[frame[n]])

                  {

                        k = frame[n];

                  }

            }

            if(flag)

            {

                  minimum\_time = 25;

                  for(n = 0; n < total\_frames; n++)

                  {

                        if(arr[frame[n]] == arr[k] && time[frame[n]] < minimum\_time)

                        {

                              temp = n;

                              minimum\_time = time[frame[n]];

                        }

                  }

                  arr[frame[temp]] = 0;

                  frame[temp] = pages[m];

            }

            for(n = 0; n < total\_frames; n++)

            {

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

            }

            printf("\n");

      }

      printf("Page Hit:\t%d\n", hit);

      return 0;

}

**OUTPUT:**

**Text

Description automatically generated**

**7. Simulate the following File allocation strategies:**

**a) Sequenced**

**b) Indexed**

**c) Linked**

**7a.)**

**Aim:** To simulate the sequenced file allocation strategy technique.

**Description:** Sequenced file allocation is one of the most used methods for allocation sequence means we allocate the block in such a manner, so that in the harddisk, all the blocks get the contiguous physical block.

**Algorithm:**

1. Start the process
2. Get the number of files
3. Get the memory requirement of the each file
4. Allocate the required locations to each in sequential order
5. Randomly select a location from available location s1=random(100);
6. Check whether the required locations are free from selected location
7. Allocate and set flag=1 to the allocated location
8. Print the results file number, length, blocks allocated
9. Stop the program

**Program:**

#include < stdio.h>

#include<conio.h>

void main()

{

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

clrscr();

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

f[i]=0;

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

x: count=0;

printf(“Enter starting block and length of files: ”);

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

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

if(f[k]==0)

count++;

if(len==count)

{

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

if(f[j]==0)

{

f[j]=1;

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

}

if(j!=(st+len-1))

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

}

else

printf(” The file is not allocated \n");

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

scanf("%d", &c);

if(c==1)

goto x;

else

exit();

getch();

}

**Output:**

Files Allocated are :

Enter starting block and length of files: 14 3

14 1

15 1

16 1

The file is allocated to disk

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

Enter starting block and length of files: 14 1

The file is not allocated

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

Enter starting block and length of files: 14 4

The file is not allocated

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

7**b.) Simulate the Indexed File allocation strategy**

**Aim:** To simulate the sequenced file allocation strategy technique.

**Algorithm:**

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

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

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

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

Empty.

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

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

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

Step 11: Terminate the process.

**Program Code:**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void main()

{

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

clrscr();

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

f[i]=0;

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

scanf("%d",&ind);

if(f[ind]!=1)

{

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

scanf("%d",&n);

}

else

{

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

goto x;

}

y: count=0;

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

{

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

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

count++;

}

if(count==n)

{

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

f[index[j]]=1;

printf("Allocated\n");

printf("File Indexed\n");

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

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

}

else

{

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

printf("Enter another file indexed");

goto y;

}

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

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**Output:**

Enter the index block: 5

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

4

1 2 3 4

Allocated

File Indexed

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

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

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

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

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

Enter the index block: 4

4 index is already allocated

Enter the index block: 6

Enter no of blocks needed and no of files for the index 6 on the disk :

2

7 8

A5llocated

File Indexed

6-------->7 : 1

6-------->8 : 1

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

**7c.) Simulate the Linked File allocation strategy**

**Aim:** To simulate the Linked File Allocationstrategy technique.

**Algorithm:**

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

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

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

Step 4: Create a stack

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

stack

Step 6: Stop the allocation.

**Program:**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void main()

{

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

clrscr();

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

f[i]=0;

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

scanf("%d", &p);

printf("Enter blocks already allocated: ");

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

{

scanf("%d", &a);

f[a]=1; }

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

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

k=len;

if(f[st]==0)

{

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

{

if(f[j]==0)

{

f[j]=1;

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

}

else

{

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

k++;

}

}

}

else

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

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

scanf("%d", &c);

if(c==1)

goto x;

else

exit(0);

getch();

}

**Output:**

Enter how many blocks already allocated: 3

Enter blocks already allocated: 1 3 5

Enter index starting block and length: 2 2

2-------->1

3 Block is already allocated

4-------->1

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