**1. Simulate the following CPU scheduling algorithms**

**a) FCFS**

**b) SJF (Premptive, Non Premptive)**

**c) Priority (Premptive, Non Premptive)**

**d) Round Robin**

**a) FCFS**

#include<stdio.h>

main()

{

int bt[20], wt[20], tat[20], i, n;

float wtavg, tatavg;

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

scanf("%d", &n);

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

{

printf("\nEnter Burst Time for Process %d -- ", i);

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

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

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

{

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

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

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

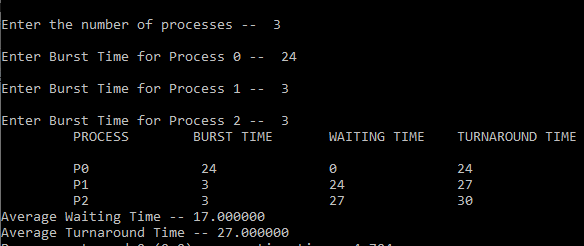
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

}

**OUTPUT**



**b) SJF**

#include<stdio.h>

main()

{

int p[20], bt[20], wt[20], tat[20], i, k, n, temp;

float wtavg, tatavg;

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

scanf("%d", &n);

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

{

p[i]=i;

printf("Enter Burst Time for Process %d -- ", i);

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

}

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

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

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

{

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=p[i];

p[i]=p[k];

p[k]=temp;

}

wt[0] = wtavg = 0;

tat[0] = tatavg = bt[0];

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

{

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

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

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

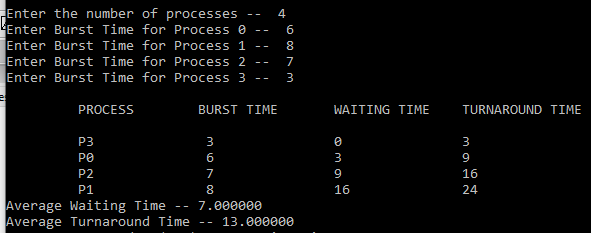
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg/n);

printf("\nAverage Turnaround Time -- %f", tatavg/n);

}

**OUTPUT**



**c) PRIORITY**

#include<stdio.h>

main()

{

int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;

float wtavg, tatavg;

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

scanf("%d",&n);

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

{

p[i] = i;

printf("Enter the Burst Time & Priority of Process %d --- ",i);

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

}

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

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

if(pri[i] > pri[k])

{

temp=p[i];

p[i]=p[k];

p[k]=temp;

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=pri[i];

pri[i]=pri[k];

pri[k]=temp;

}

wtavg = wt[0] = 0;

tatavg = tat[0] = bt[0];

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

{

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

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

wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");

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

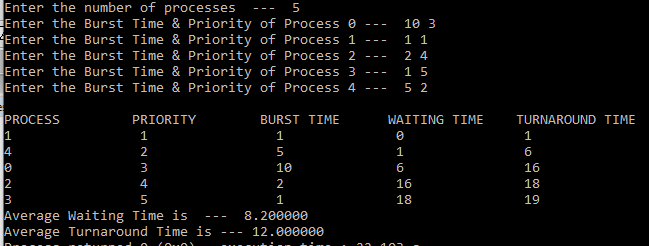
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);

printf("\nAverage Waiting Time is --- %f",wtavg/n);

printf("\nAverage Turnaround Time is --- %f",tatavg/n);

}

**OUTPUT**



**d) ROUND ROBIN**

#include<stdio.h>

main()

{

int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;

float awt=0,att=0,temp=0;

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

scanf("%d",&n);

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

{

printf("\nEnter Burst Time for process %d -- ", i+1);

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

ct[i]=bu[i];

}

printf("\nEnter the size of time slice -- ");

scanf("%d",&t);

max=bu[0];

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

if(max<bu[i])

max=bu[i];

for(j=0;j<(max/t)+1;j++)

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

if(bu[i]!=0)

if(bu[i]<=t)

{

tat[i]=temp+bu[i];

temp=temp+bu[i];

bu[i]=0;

}

else

{

bu[i]=bu[i]-t;

temp=temp+t;

}

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

{

wa[i]=tat[i]-ct[i];

att+=tat[i];

awt+=wa[i];

}

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

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

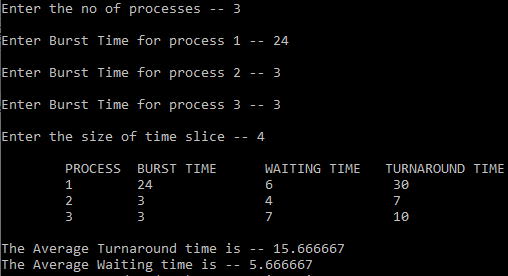
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);

printf("\nThe Average Turnaround time is -- %f",att/n);

printf("\nThe Average Waiting time is -- %f ",awt/n);

}

**OUTPUT**



**e) SJF(PREEMPTION)**

#include<stdio.h>

struct process

{

int WT,AT,BT,TAT;

};

struct process a[10];

int main()

{

int n,temp[10];

int count=0,t=0,short\_P;

float total\_WT=0, total\_TAT=0,Avg\_WT,Avg\_TAT;

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

scanf("%d",&n);

printf("Enter the arrival time and burst time of the process\n");

printf("AT BT\n");

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

{

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

temp[i]=a[i].BT; // copying the burst time in a temp array for the further use in calculation of WT

}

a[9].BT=10000; // we initialize the burst time of a process with the maximum

for(t=0;count!=n;t++) // loop will be execute until all the process

{ // complete so we use count!= number of the process

short\_P=9; // for finding min burst it is useful

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

{

if(a[i].BT<a[short\_P].BT && (a[i].AT<=t && a[i].BT>0))

{

short\_P=i;

}

}

a[short\_P].BT=a[short\_P].BT-1;

if(a[short\_P].BT==0) // if any process is completed

{

count++; // one process complete

a[short\_P].WT=t+1-a[short\_P].AT-temp[short\_P];

a[short\_P].TAT=t+1-a[short\_P].AT;

total\_WT=total\_WT+a[short\_P].WT; // total calculation

total\_TAT=total\_TAT+a[short\_P].TAT;

}

}

Avg\_WT=total\_WT/n;

Avg\_TAT=total\_TAT/n;

printf("P\_Id\tWT\tTAT\n"); // printing of the answer

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

{

printf("%d\t%d\t%d\n",i+1,a[i].WT,a[i].TAT);

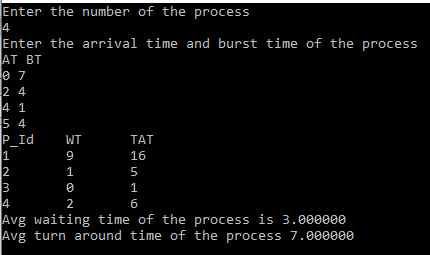
}

printf("Avg waiting time of the process is %f\n",Avg\_WT);

printf("Avg turn around time of the process %f\n",Avg\_TAT);

}

**OUTPUT**



**f) PRIORITY((PREEMPTION)**

#include<stdio.h>

struct process

{

int WT,AT,BT,TAT,PT;

};

struct process a[10];

int main()

{

int n,temp[10],t,count=0,short\_p;

float total\_WT=0,total\_TAT=0,Avg\_WT,Avg\_TAT;

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

scanf("%d",&n);

printf("Enter the arrival time , burst time and priority of the process\n");

printf("AT BT PT\n");

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

{

scanf("%d%d%d",&a[i].AT,&a[i].BT,&a[i].PT);

temp[i]=a[i].BT; // copying the burst time in a temp array for the further use

}

a[9].PT=10000; // we initialize the burst time of a process with maximum

for(t=0;count!=n;t++)

{

short\_p=9;

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

{

if(a[short\_p].PT>a[i].PT && a[i].AT<=t && a[i].BT>0)

{

short\_p=i;

}

}

a[short\_p].BT=a[short\_p].BT-1; // if any process is completed

if(a[short\_p].BT==0)

{

count++; // one process is completed, so count increases by 1

a[short\_p].WT=t+1-a[short\_p].AT-temp[short\_p];

a[short\_p].TAT=t+1-a[short\_p].AT;

total\_WT=total\_WT+a[short\_p].WT; // total calculation

total\_TAT=total\_TAT+a[short\_p].TAT;

}

}

Avg\_WT=total\_WT/n;

Avg\_TAT=total\_TAT/n;

printf("ID WT TAT\n"); // printing of the answer

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

{

printf("%d %d\t%d\n",i+1,a[i].WT,a[i].TAT);

}

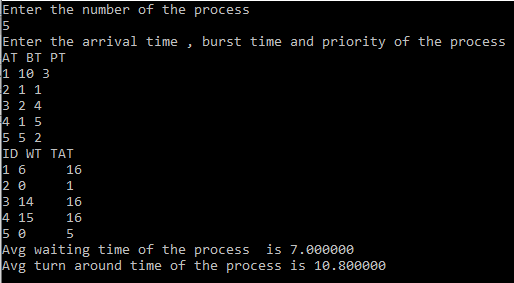
printf("Avg waiting time of the process is %f\n",Avg\_WT);

printf("Avg turn around time of the process is %f\n",Avg\_TAT);

return 0;

}

**OUTPUT**



**2. Simulate the following Process Synchronization techniques**

**a) Bounded-Buffer problem**

**b) Readers-Writers problem**

**c) Dining philosophers problem using semaphores**

**a) Bounded-Buffer problem**

#include <stdio.h>

#include <stdlib.h>

int mutex = 1, full = 0, 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");

    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;

        // Exit Condition

        case 3:

            exit(0);

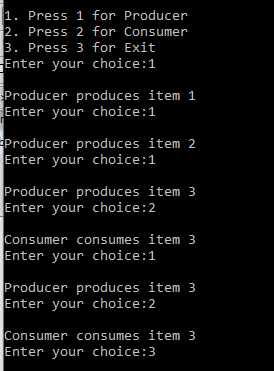
            break;

        }

    }

}

**OUTPUT**



**b) Readers-Writers problem**

#include<semaphore.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

sem\_t x,y;

pthread\_t tid;

pthread\_t writerthreads[100],readerthreads[100];

int readercount = 0;

void \*reader(void\* param)

{

sem\_wait(&x);

readercount++;

if(readercount==1)

sem\_wait(&y);

sem\_post(&x);

printf("%d reader is inside\n",readercount);

usleep(3);

sem\_wait(&x);

readercount--;

if(readercount==0)

{

sem\_post(&y);

}

sem\_post(&x);

printf("%d Reader is leaving\n",readercount+1);

return NULL;

}

void \*writer(void\* param)

{

printf("Writer is trying to enter\n");

sem\_wait(&y);

printf("Writer has entered\n");

sem\_post(&y);

printf("Writer is leaving\n");

return NULL;

}

int main()

{

int n2,i;

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

scanf("%d",&n2);

printf("\n");

int n1[n2];

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

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

for(i=0;i<n2;i++)

{

pthread\_create(&writerthreads[i],NULL,reader,NULL);

pthread\_create(&readerthreads[i],NULL,writer,NULL);

}

for(i=0;i<n2;i++)

{

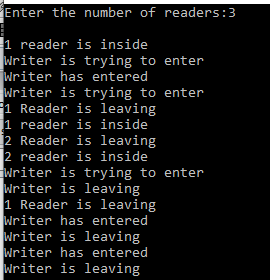
pthread\_join(writerthreads[i],NULL);

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

}

}

**OUTPUT**



**c) Dining philosophers problem using semaphores**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

sem\_t mutex;

sem\_t S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating

state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is Eating\n", phnum + 1);

sem\_post(&S[phnum]);

}

}

// take up chopsticks

void take\_fork(int phnum)

{

sem\_wait(&mutex);

// state that hungry

state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating

test(phnum);

sem\_post(&mutex);

// if unable to eat wait to be signalled

sem\_wait(&S[phnum]);

sleep(1);

}

// put down chopsticks

void put\_fork(int phnum)

{

sem\_wait(&mutex);

// state that thinking

state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philosopher(void\* num)

{

while (1)

{

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main()

{

int i;

pthread\_t thread\_id[N];

// initialize the semaphores

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

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

sem\_init(&S[i], 0, 0);

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

{

// create philosopher processes

pthread\_create(&thread\_id[i], NULL,philosopher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);

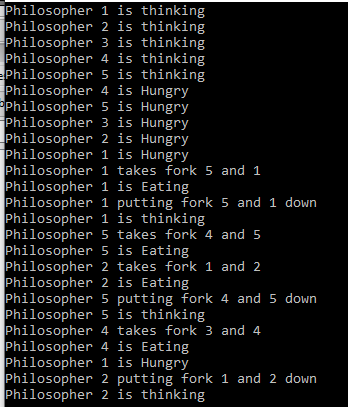
}

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

pthread\_join(thread\_id[i], NULL);

}

**OUTPUT**



**d) Dining-Philosophers Solution using Monitors**

3. Simulate the following

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

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

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

**Program**

//MFT

#include<stdio.h>

main()

{

int ms,bs,nob,ef,n,mp[10],tif=0;

int i,p=0;

printf("enter total memory:");

scanf("%d",&ms);

printf("enter memory size:");

scanf("%d",&bs);

nob=ms/bs;

ef=ms-nob\*bs;

printf("enter total processes:");

scanf("%d",&n);

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

{

printf("enter total memory for process %d:",i+1);

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

}

printf("\nno of blocks available %d",nob);

printf("\n\nprocess\tmemory required\t\tallocated\tinternal fragmentation");

for(i=0;i<n&&p<nob;i++)

{

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

if(mp[i]>bs)

printf("\t\tno\t");

else

{

printf("\t\tyes\t%d",bs-mp[i]);

tif=tif+bs-mp[i];

p++;

}

}

if(i<n)

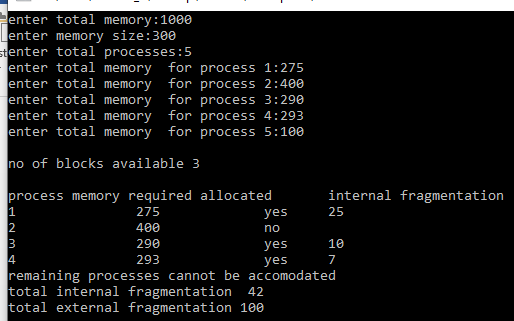
printf("\nremaining processes cannot be accomodated");

printf("\ntotal internal fragmentation %d",tif);

printf("\ntotal external fragmentation %d\n",ef);

}

**Output**



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

**Program**

#include<stdio.h>

main()

{

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

char ch = 'y';

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++)

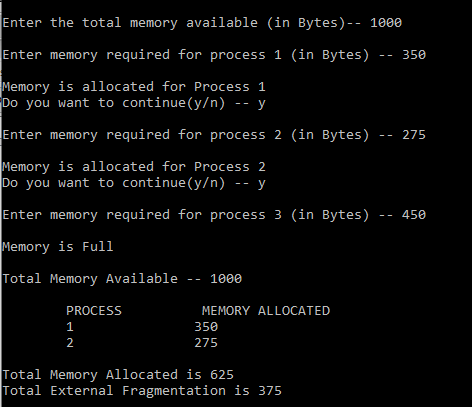
printf("\n \t%d\t\t%d",i+1,mp[i]);

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

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

}

**Output**



4.Simulate the following File allocation strategies

a) Contiguous

b) Linked

c) Indexed

a) Contiguous / sequenced

**Program**

#include<stdio.h>

#include<stdlib.h>

main()

{

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

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

f[i]=0;

X:

printf("\n Enter the starting block & length of file");

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

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

if(f[j]==0)

{

f[j]=1;

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

}

else

{

printf("Block already allocated");

break;

}

if(j==(st+len))

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

printf("\n if u want to enter more files?(y-1/n-0)");

scanf("%d",&c);

if(c==1)

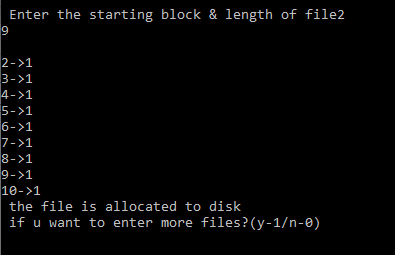
goto X;

else

exit(0);

}

**OUTPUT**



b) Linked

#include<stdio.h>

#include<stdlib.h>

void main()

{

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

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

f[i]=0;

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

scanf("%d",&p);

printf("Enter blocks already allocated: ");

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

{

scanf("%d",&a);

f[a]=1;

}

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

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

k=len;

if(f[st]==0)

{

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

{

if(f[j]==0)

{

f[j]=1;

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

}

else

{

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

k++;

}

}

}

else

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

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

scanf("%d", &c);

if(c==1)

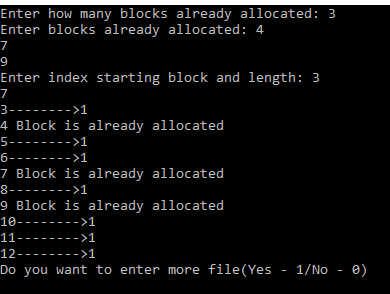
goto x;

else

exit(0);

}

**OUTPUT**



c) Indexed

**Program**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int main()

{

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

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

f[i]=0;

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

scanf("%d",&ind);

if(f[ind]!=1)

{

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

scanf("%d",&n);

}

else

{

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

goto x;

}

y: count=0;

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

{

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

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

count++;

}

if(count==n)

{

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

f[index[j]]=1;

printf("Allocated\n");

printf("File Indexed\n");

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

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

}

else

{

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

printf("Enter another file indexed");

goto y;

}

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

scanf("%d", &c);

if(c==1)

goto x;

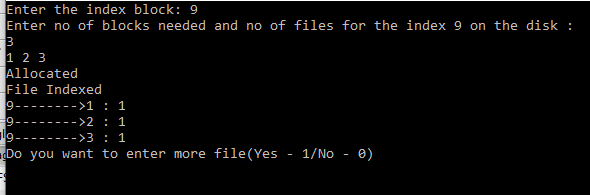
else

exit(0);

getch();

}

**OUTPUT**



**5**. Simulate Bankers Algorithm for Dead Lock Avoidance

#include<stdio.h>

#include<conio.h>

int max[100][100],alloc[100][100],need[100][100],avail[100];

int n,r;

void input();

void show();

void cal();

int main()

{

int i,j;

printf("\*\*\*\*\*\*\*\*\*\* Banker's Algorithm \*\*\*\*\*\*\*\*\*\*\*\*\n");

input();

show();

cal();

getch();

return 0;

}

void input()

{

int i,j;

printf("Enter the no of Processes\t");

scanf("%d",&n);

printf("Enter the no of resources instances\t");

scanf("%d",&r);

printf("Enter the Max Matrix\n");

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

{

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

{

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

}

}

printf("Enter the Allocation Matrix\n");

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

{

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

{

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

}

}

printf("Enter the available Resources\n");

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

{

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

}

}

void show()

{

int i,j;

printf("Process\t Allocation\t Max\t Available\t");

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

{

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

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

{

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

}

printf("\t");

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

{

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

}

printf("\t");

if(i==0)

{

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

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

}

}

}

void cal()

{

int finish[100],temp,need[100][100],flag=1,k,c1=0;

int safe[100];

int i,j;

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

{

finish[i]=0;

}

//find need matrix

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

{

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

{

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

}

}

printf("\n");

while(flag)

{

flag=0;

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

{

int c=0;

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

{

if((finish[i]==0)&&(need[i][j]<=avail[j]))

{

c++;

if(c==r)

{

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

{

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

finish[i]=1;

flag=1;

}

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

if(finish[i]==1)

{

i=n;

}

}

}

}

}

}

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

{

if(finish[i]==1)

{

c1++;

}

else

{

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

}

}

if(c1==n)

{

printf("\n The system is in safe state");

}

else

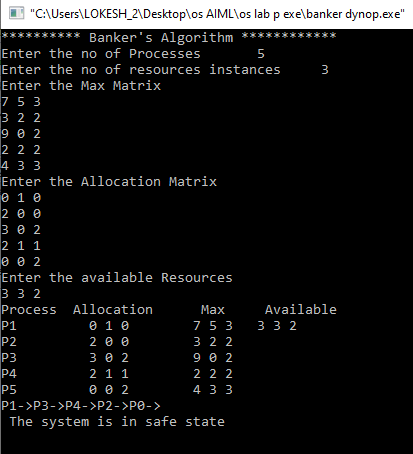
{

printf("\n Process are in dead lock");

printf("\n System is in unsafe state");

}

}



**deadlock prevention**

#include<stdio.h>

#include<conio.h>

int max[10][10],alloc[10][10],need[10][10],avail[10],i,j,p,r,finish[10]={0},flag=0;

main( )

{

printf("\n\nSIMULATION OF DEADLOCK PREVENTION");

printf("Enter no. of processes, resources");

scanf("%d%d",&p,&r);

printf("Enter allocation matrix");

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

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

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

printf("enter max matrix");

for(i=0;i<p;i++) /\*reading the maximum matrix and availale

matrix\*/

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

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

printf("enter available matrix");

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

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

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

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

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

fun(); /\*calling function\*/

if(flag==0)

{

if(finish[i]!=1)

{

printf("\n\n Failing :Mutual exclusion");

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

{ /\*checking for mutual exclusion\*/

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

avail[j]=need[i][j];

}

fun();

printf("\n By allocating required resources to process %d dead lock is prevented" ,i);

printf("\n\n lack of no preemption");

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

{

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

avail[j]=need[i][j];

alloc[i][j]=0;

}

fun( );

printf("\n\n dead lock is prevented by allocating needed resources");

printf(" \n \n failing:Hold and Wait condition ");

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

{ /\*checking hold and wait condition\*/

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

avail[j]=need[i][j];

}

fun( );

printf("\n AVOIDING ANY ONE OF THE CONDITION, U CAN PREVEMT DEADLOCK");

}

}

getch( );

}

fun( )

{

while(1)

{

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

{

if(finish[i]==0)

{

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

{

if(need[i][j]<=avail[j])

continue;

else

break;

}

if(j==r)

{

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

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

flag=1;

finish[i]=1;

}

}

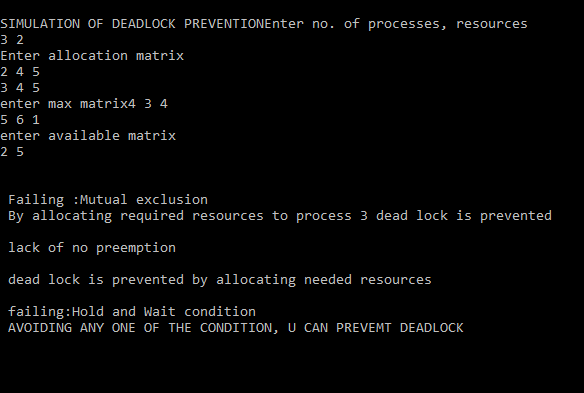
}

if(flag==0)

break;

}

}



6. Simulate the following page replacement algorithms.

a) FIFO

b) LRU

c) LFU

d) MFU

1. **FIFO**

#include<stdio.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");

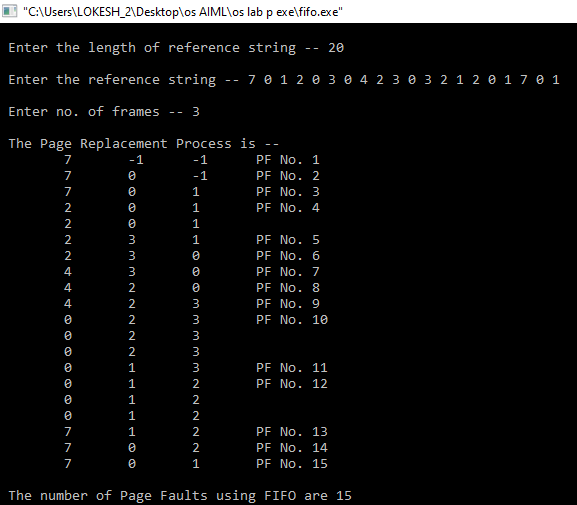
if(count==f)

count=0;

}

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

}



1. LRU

#include<stdio.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)

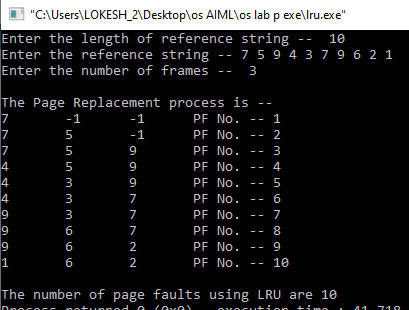
printf("PF No. -- %d" , pf);

printf("\n");

}

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

}



**C)LFU**

#include<stdio.h>

int main()

{

int rs[30],st[10],ru[10],place\_rank[10],n,f,i,j,k,pf=0,min=0,ind,count=0;

printf("enter no of references:");

scanf("%d",&n);

printf("enter reference string:");

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

{

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

}

printf("enter no of frames:");

scanf("%d",&f);

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

{

st[j]=-1;

ru[j]=0;

place\_rank[j]=0;

}

printf("page replacement process:\n");

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

{

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

{

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

{

ru[k]++;

break;

}

}

if(k==f)

{

min=0;

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

if(ru[k]<ru[min])

min=k;

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

{

if(ru[j]==ru[min] && place\_rank[j]<place\_rank[min])

{

min=j;

}

}

st[min]=rs[i];

place\_rank[min]=++count;

ru[min]=1;

pf++;

}

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

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

if(k==f)

printf("PF no.%d",pf);

printf("\n");

}

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

}

|  |
| --- |
|  |
|  |
|  |  |
|  |  |
|  |  |
|  |  |
| 1. **MFU**   #include<stdio.h>  #include<stdlib.h>  void main()  {  int hit=0,miss=0,i,j,noPages,noFrames,min;  int frames[10],pages[20];  int flag=0,flag1=0,flag2=0;  int flagFound=0;  int count=0;  int frameAge[50],frameFREQ[50];  printf("enter number of frames\n");  scanf("%d",&noFrames);  printf("enter number of pages\n");  scanf("%d",&noPages);  printf("enter the page string ");  for(i=0;i<noPages;i++)  {  scanf("%d",&pages[i]);  }  for(i=0;i<noFrames;i++)  {  frames[i]=-1;  frameAge[i]=-1;  }  for(j=0;j<noFrames;j++)  frameFREQ[j]=0;  for(j=0;j<noPages;j++)  {  int index;  printf(" page:%d ",pages[j] );  flagFound=0,flag=0,flag2=0;  for(i=0;i<noFrames;i++)  {  if(frames[i]==pages[j])  {  flagFound=1;  flag=1;  index=i;  printf("hit ");  hit++;  break;  }  }  if(flagFound==0) //if frame not found and empty frame avalible  {  for(i=0;i<noFrames;i++)  {  if(frames[i]==-1)  {  frames[i]=pages[j];  flag=1;  count++;  frameAge[i]=count;  miss++;  frameFREQ[i]=1;  printf("miss ");  break;  } }  if(flag==0)  {  int bestmfu=0;  for(i=0;i<noFrames;i++)  {  if(frameFREQ[i]>frameFREQ[bestmfu])  bestmfu=i;  }  frames[bestmfu]=pages[j];  miss++;  printf("miss ");  frameFREQ[bestmfu]=1;  }  } //FLAG FOUND ends  else  {  frameFREQ[index]++;  }  for(i=0;i<noFrames;i++)  {  printf(" %d ",frames[i] );  }  printf("\n");  }  printf("number of hits %d\n",hit );  printf("number of miss %d\n",miss );  } |  |
|  |  |
|  |  |
|  |  |
| 7.Simulate the following disk-scheduling algorithms  a) FCFS  b) SSTF  c) SCAN  d) C-SCAN  e) LOOK  f) C-LOOK  **a) FCFS** |  |
| #include<stdio.h>  #include<stdlib.h>  int main()  {  int RQ[100],i,n,TotalHeadMoment=0,initial;  printf("Enter the number of Requests\n");  scanf("%d",&n);  printf("Enter the Requests sequence\n");  for(i=0;i<n;i++)  scanf("%d",&RQ[i]);  printf("Enter initial head position\n");  scanf("%d",&initial);  // logic for FCFS disk scheduling  for(i=0;i<n;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  printf("Total head moment is %d",TotalHeadMoment);  return 0;  } |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **b) SSTF**  #include<stdio.h>  #include<stdlib.h>  int main()  {  int RQ[100],i,n,TotalHeadMoment=0,initial,count=0;  printf("Enter the number of Requests\n");  scanf("%d",&n);  printf("Enter the Requests sequence\n");  for(i=0;i<n;i++)  scanf("%d",&RQ[i]);  printf("Enter initial head position\n");  scanf("%d",&initial);  // logic for sstf disk scheduling  /\* loop will execute until all process is completed\*/  while(count!=n)  {  int min=1000,d,index;  for(i=0;i<n;i++)  {  d=abs(RQ[i]-initial);  if(min>d)  {  min=d;  index=i;  }  }  TotalHeadMoment=TotalHeadMoment+min;  initial=RQ[index];  // 1000 is for max  // you can use any number  RQ[index]=1000;  count++;  }  printf("Total head movement is %d",TotalHeadMoment);  return 0;  } |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **c) SCAN**  #include<stdio.h>  #include<stdlib.h>  int main()  {  int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;  printf("Enter the number of Requests\n");  scanf("%d",&n);  printf("Enter the Requests sequence\n");  for(i=0;i<n;i++)  scanf("%d",&RQ[i]);  printf("Enter initial head position\n");  scanf("%d",&initial);  printf("Enter total disk size\n");  scanf("%d",&size);  printf("Enter the head movement direction for high 1 and for low 0\n");  scanf("%d",&move);  // logic for Scan disk scheduling  /\*logic for sort the request array \*/  for(i=0;i<n;i++)  {  for(j=0;j<n-i-1;j++)  {  if(RQ[j]>RQ[j+1])  {  int temp;  temp=RQ[j];  RQ[j]=RQ[j+1];  RQ[j+1]=temp;  }  }  }  int index;  for(i=0;i<n;i++)  {  if(initial<RQ[i])  {  index=i;  break;  }  }  // if movement is towards high value  if(move==1)  {  for(i=index;i<n;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  // last movement for max size  TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);  initial = size-1;  for(i=index-1;i>=0;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  // if movement is towards low value  else  {  for(i=index-1;i>=0;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  // last movement for min size  TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);  initial =0;  for(i=index;i<n;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  printf("Total head movement is %d",TotalHeadMoment);  return 0;  } |  |
|  |  |
|  |  |
|  |  |
|  |  |
| 1. **C-SCAN**   #include<stdio.h>  #include<stdlib.h>  int main()  {  int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;  printf("Enter the number of Requests\n");  scanf("%d",&n);  printf("Enter the Requests sequence\n");  for(i=0;i<n;i++)  scanf("%d",&RQ[i]);  printf("Enter initial head position\n");  scanf("%d",&initial);  printf("Enter total disk size\n");  scanf("%d",&size);  printf("Enter the head movement direction for high 1 and for low 0\n");  scanf("%d",&move);  // logic for C-Scan disk scheduling  /\*logic for sort the request array \*/  for(i=0;i<n;i++)  {  for( j=0;j<n-i-1;j++)  {  if(RQ[j]>RQ[j+1])  {  int temp;  temp=RQ[j];  RQ[j]=RQ[j+1];  RQ[j+1]=temp;  }  }  }  int index;  for(i=0;i<n;i++)  {  if(initial<RQ[i])  {  index=i;  break;  }  }  // if movement is towards high value  if(move==1)  {  for(i=index;i<n;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  // last movement for max size  TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);  /\*movement max to min disk \*/  TotalHeadMoment=TotalHeadMoment+abs(size-1-0);  initial=0;  for( i=0;i<index;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  // if movement is towards low value  else  {  for(i=index-1;i>=0;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  // last movement for min size  TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);  /\*movement min to max disk \*/  TotalHeadMoment=TotalHeadMoment+abs(size-1-0);  initial =size-1;  for(i=n-1;i>=index;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  printf("Total head movement is %d",TotalHeadMoment);  return 0;  }    **e) LOOK**  #include<stdio.h>  #include<stdlib.h>  int main()  {  int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;  printf("Enter the number of Requests\n");  scanf("%d",&n);  printf("Enter the Requests sequence\n");  for(i=0;i<n;i++)  scanf("%d",&RQ[i]);  printf("Enter initial head position\n");  scanf("%d",&initial);  printf("Enter total disk size\n");  scanf("%d",&size);  printf("Enter the head movement direction for high 1 and for low 0\n");  scanf("%d",&move);  // logic for look disk scheduling  /\*logic for sort the request array \*/  for(i=0;i<n;i++)  {  for(j=0;j<n-i-1;j++)  {  if(RQ[j]>RQ[j+1])  {  int temp;  temp=RQ[j];  RQ[j]=RQ[j+1];  RQ[j+1]=temp;  }  }  }  int index;  for(i=0;i<n;i++)  {  if(initial<RQ[i])  {  index=i;  break;  }  }  // if movement is towards high value  if(move==1)  {  for(i=index;i<n;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  for(i=index-1;i>=0;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  // if movement is towards low value  else  {  for(i=index-1;i>=0;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  for(i=index;i<n;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  printf("Total head movement is %d",TotalHeadMoment);  return 0;  } |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **f) C-LOOK** |  |
| #include<stdio.h>  #include<stdlib.h>  int main()  {  int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;  printf("Enter the number of Requests\n");  scanf("%d",&n);  printf("Enter the Requests sequence\n");  for(i=0;i<n;i++)  scanf("%d",&RQ[i]);  printf("Enter initial head position\n");  scanf("%d",&initial);  printf("Enter total disk size\n");  scanf("%d",&size);  printf("Enter the head movement direction for high 1 and for low 0\n");  scanf("%d",&move);  // logic for C-look disk scheduling  /\*logic for sort the request array \*/  for(i=0;i<n;i++)  {  for( j=0;j<n-i-1;j++)  {  if(RQ[j]>RQ[j+1])  {  int temp;  temp=RQ[j];  RQ[j]=RQ[j+1];  RQ[j+1]=temp;  }  }  }  int index;  for(i=0;i<n;i++)  {  if(initial<RQ[i])  {  index=i;  break;  }  }  // if movement is towards high value  if(move==1)  {  for(i=index;i<n;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  for( i=0;i<index;i++)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  // if movement is towards low value  else  {  for(i=index-1;i>=0;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  for(i=n-1;i>=index;i--)  {  TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);  initial=RQ[i];  }  }  printf("Total head movement is %d",TotalHeadMoment);  return 0;  } |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Lab frequently asked questions**

**EX1**

1. What is the advantage of round robin CPU scheduling algorithm?

2. Which CPU scheduling algorithm is for real-time operating system?

3. In general, which CPU scheduling algorithm works with highest waiting time?

4. Is it possible to use optimal CPU scheduling algorithm in practice?

5. What is the real difficulty with the SJF CPU scheduling algorithm?

**EX linked indexed**

1. Identify ideal scenarios where sequential, indexed and linked file allocation strategies are most appropriate?

2. What are the disadvantages of sequential file allocation strategy?

3. What is an index block?

4. What is the file allocation strategy used in UNIX?

**EX mft mvt**

1. What is the purpose of memory management unit?

2. Differentiate between logical address and physical address?

3. What are the different types of address binding techniques?

4. What is the basic idea behind contiguous memory allocation?

5. How is dynamic memory allocation useful in multiprogramming operating systems?

**EX deadlock**

1. How can be the resource allocation graph used to identify a deadlock situation?

2. How is Banker’s algorithm useful over resource allocation graph technique?

3. Differentiate between deadlock avoidance and deadlock prevention?

**EX page replacement**

1. Define the concept of virtual memory?

2. What is the purpose of page replacement?

3. Define the general process of page replacement?

4. List out the various page replacement techniques?

5. What is page fault?

**EX semaphores**

1. What is the need for process synchronization?

2. Define a semaphore?

3. Define producer-consumer problem?

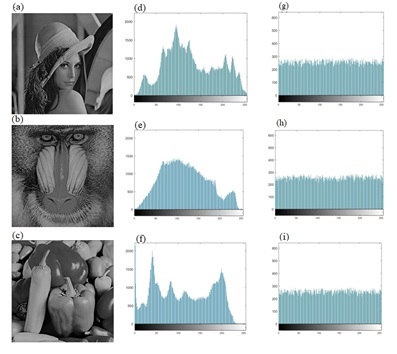
4. Discuss the consequences of considering bounded and unbounded buffers in producer- consumer problem?

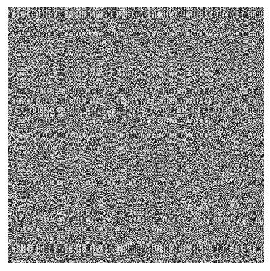
**EX disk scheduling**

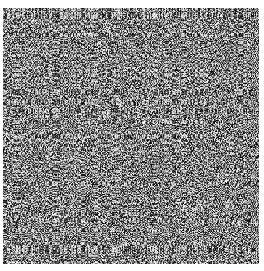
1. Define the terms – disks seek time, disk access time and rotational latency

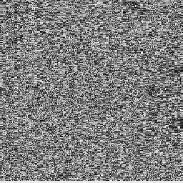
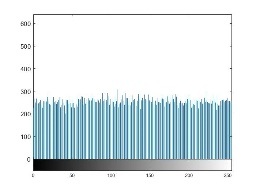
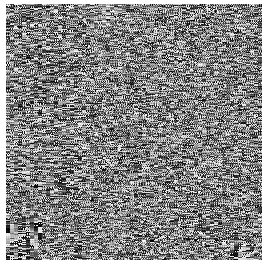
2. What is the advantage of C-SCAN algorithm over SCAN algorithm?

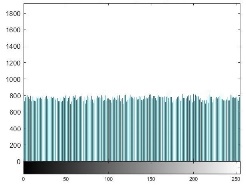
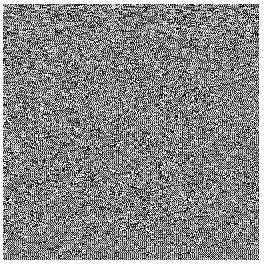
3. Which disk scheduling algorithm has highest rotational latency? Why?

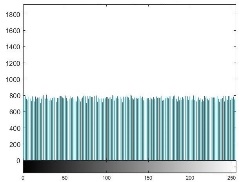
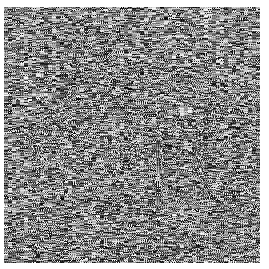
****

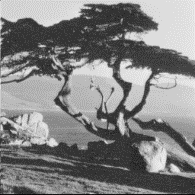




  
 C:\Users\LOKESH_2\Desktop\images jedt\4.1.01 Female.tif 

 C:\Users\LOKESH_2\Desktop\images jedt\couple.tif 

 C:\Users\LOKESH_2\Desktop\images jedt\house.tif  

 C:\Users\LOKESH_2\Desktop\images jedt\house.tif