**PRACTICAL – 1**

**AIM :**

Implement shell scripting making a script to take a file name, start line and end line number from user and print text of file accordingly.

**THEORY:**

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension eg. myscript.sh

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

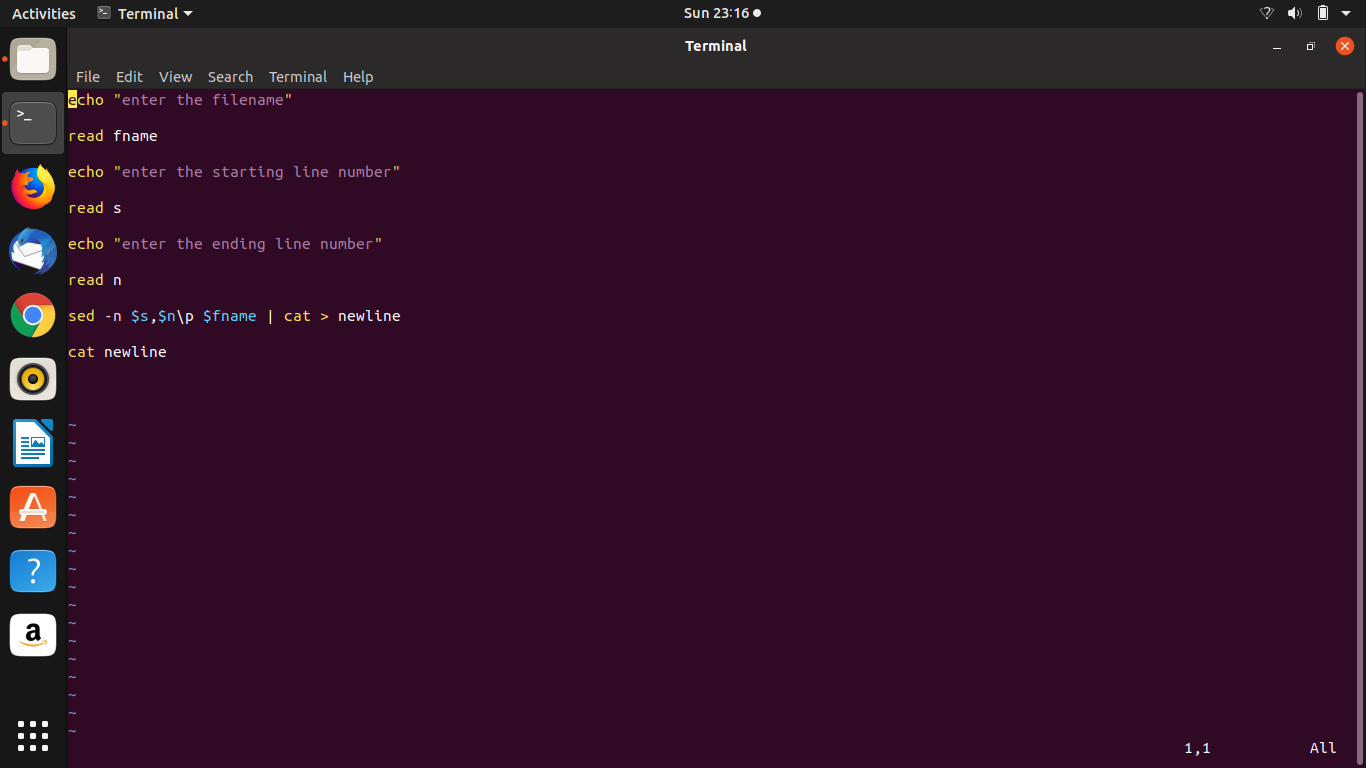
Shell Keywords – if, else, break etc.

Shell commands – cd, ls, echo, pwd, touch etc.

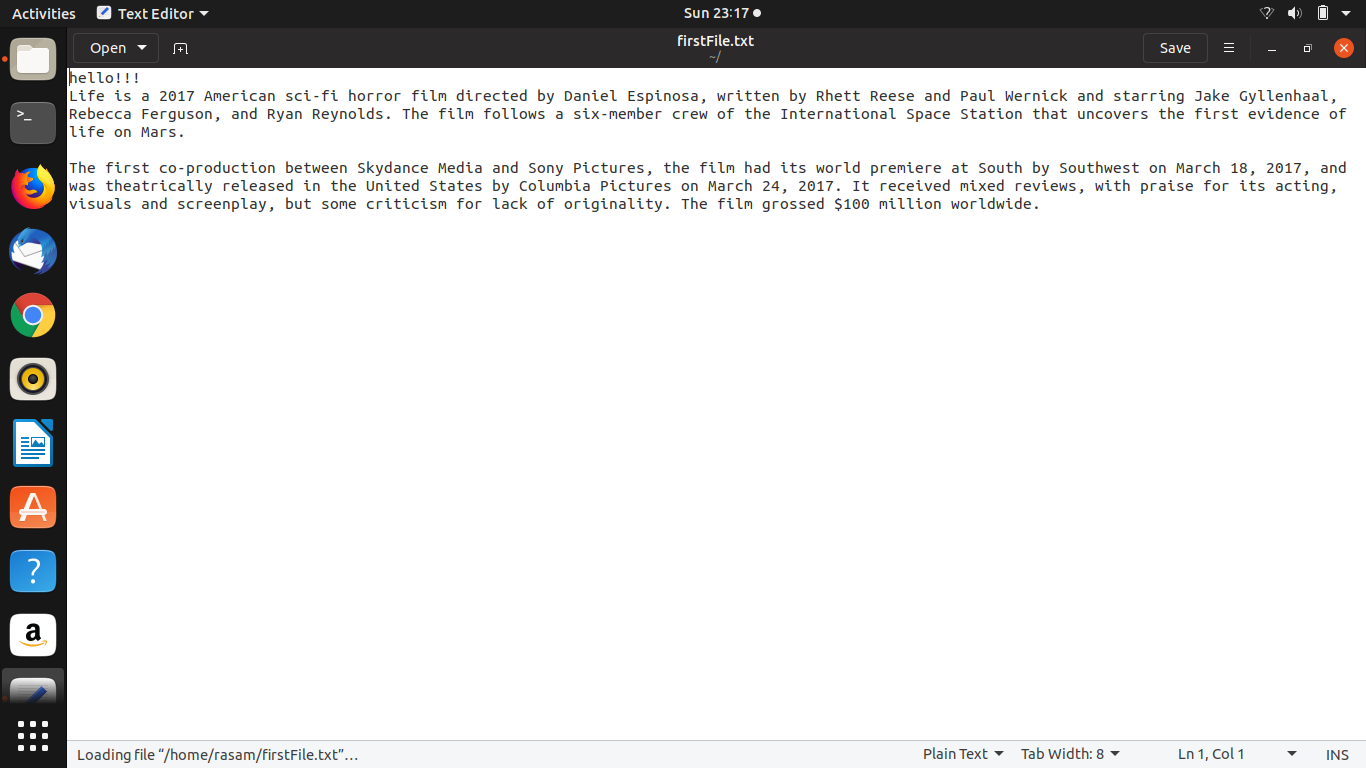
Functions

Control flow – if..then..else, case and shell loops etc.

**Shell Script:**

****

**Actual File:**

****

**OUTPUT:**



**PRACTICAL-2**

**Aim:**

Write a program for first come first serve CPU scheduling.

**Theory:**

Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-preemptive scheduling algorithm.

**Algorithm:**

1. Input the processes along with their burst time (bt).
2. Find waiting time (wt) for all processes.
3. As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.
4. Find waiting time for all other processes i.e. for process i ->

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

1. Find turnaround time = waiting\_time + burst\_time for all processes.
2. Find average waiting time =
   * 1. total\_waiting\_time / no\_of\_processes.
3. Similarly, find average turnaround time =
   * 1. total\_turn\_around\_time / no\_of\_processes.

**Program:**

#include<bits/stdc++.h>

using namespace std;

struct process

{

int art,bt,wt,tat,id,ct;

};

bool compare(process p1,process p2)

{

return p1.art<p2.art;

}

void calculate\_wt(process p[],int n)

{

int service\_time = 0;

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

{

p[i].wt = service\_time-p[i].art;

service\_time +=p[i].bt;

p[i].ct=service\_time;

if (p[i].wt < 0)

p[i].wt = 0;

}

}

void calculate\_tat(process p[],int n)

{

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

{

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

}

}

int main()

{

cout<<"enter no. of processes : ";

int n;

cin>>n;

process p[n];

cout<<"enter the arrival time and burst time of each process : ";

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

{

cin>>p[i].art>>p[i].bt;

p[i].id = i+1;

}

sort(p,p+n,compare);

calculate\_wt(p,n);

calculate\_tat(p,n);

cout<<setw(4)<<"AT"<<setw(4)<<"BT"<<setw(4)<<"CT"<<setw(4)<<"TAT"<<setw(4)<<"WT\n";

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

{

cout<<setw(4)<<p[i].art<<setw(4)<<p[i].bt<<setw(4)<<p[i].ct<<setw(4)<<p[i].tat<<setw(4)<<p[i].wt<<"\n";

}

float atat=0;

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

{

atat+=p[i].tat;

}

cout<<"average turn around time: "<<atat/n<<"\n";

float awt;

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

{

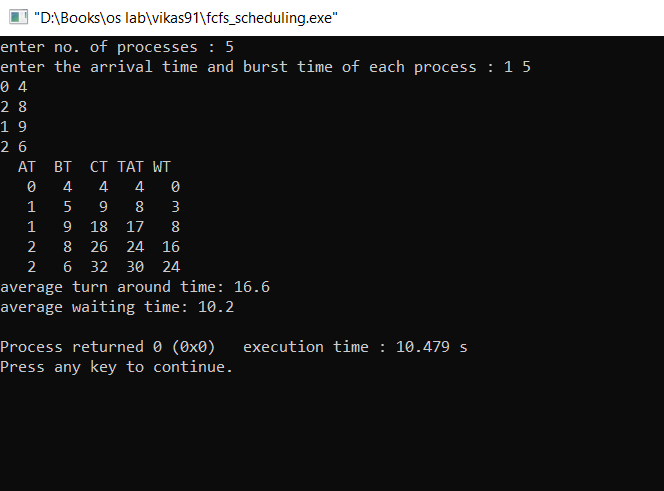
awt+=p[i].wt;

}

cout<<"average waiting time: "<<awt/n<<"\n";

}

**Output:**

****

**PRACTICAL-3**

**Aim:**

Write a program for shortest job first CPU scheduling.

**Theory:**

Process which have the shortest burst time are scheduled first.If two processes have the same bust time then FCFS is used to break the tie. It is a non-preemptive scheduling algorithm.

**Algorithm:**

1. Sort all the process according to the arrival time.
2. Then select that process which has minimum arrival time and minimum Burst time.
3. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

**Program:**

#include<bits/stdc++.h>

using namespace std;

struct process

{

int id,art,bt,ct,tat,wt,vis;

};

bool comp(process x, process y)

{

return x.art<y.art;

}

struct comp\_burst

{

bool operator()(process p1, process p2)

{

if(p1.bt==p2.bt)

return p1.id>p2.id;

return p1.bt>p2.bt;

}

};

main()

{

int n;

cout<<"enter no. of processes: ";

cin>>n;

process p[n];

cout<<"enter arrival time and burst time for each process\n";

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

{

p[i].id=i;

p[i].vis=0;

cin>>p[i].art>>p[i].bt;

}

sort(p,p+n,comp);

priority\_queue <process, vector<process>, comp\_burst > pq;

int time = p[0].art;

pq.push(p[0]);

p[0].vis=1;

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

{

if(p[i].art<=time && p[i].vis==0)

{

pq.push(p[i]);

p[i].vis=1;

}

else if(p[i].art>time)

break;

}

while(p[n-1].vis==0 || !pq.empty())

{

while(!pq.empty())

{

process pr = pq.top();

pq.pop();

time+=pr.bt;

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

{

if(p[i].id==pr.id)

{

p[i].ct=time;

break;

}

}

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

{

if(p[i].art<=time && p[i].vis==0)

{

pq.push(p[i]);

p[i].vis=1;

}

else if(p[i].art>time)

break;

}

}

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

{

if(p[i].vis==0)

{

time=p[i].art;

break;

}

}

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

{

if(p[i].art<=time && p[i].vis==0)

{

pq.push(p[i]);

p[i].vis=1;

}

else if(p[i].art>time)

break;

}

}

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

{

p[i].wt=p[i].ct-p[i].bt-p[i].art;

p[i].tat=p[i].ct-p[i].art;

}

cout<<setw(4)<<"AT"<<setw(4)<<"BT"<<setw(4)<<"CT"<<setw(4)<<"TAT"<<setw(4)<<"WT\n";

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

{

cout<<setw(4)<<p[i].art<<setw(4)<<p[i].bt<<setw(4)<<p[i].ct<<setw(4)<<p[i].tat<<setw(4)<<p[i].wt<<"\n";

}

float atat=0;

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

{

atat+=p[i].tat;

}

cout<<"average turn around time: "<<atat/5<<"\n";

float awt;

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

{

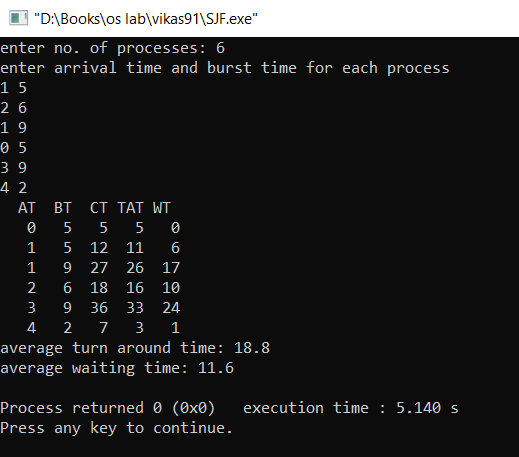
awt+=p[i].wt;

}

cout<<"average waiting time: "<<awt/5<<"\n";

}

**Output:**



**PRACTICAL 4**

**Aim:**

Write a program for round robin CPU scheduling.

**Theory:**

Each process is assigned a fixed time(Time Quantum/Time Slice) in cyclic way.It is designed especially for the time-sharing system. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. To implement Round Robin scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

**Algorithm:**

1- Create an array rem\_bt[] to keep track of remaining

burst time of processes. This array is initially a

copy of bt[] (burst times array)

2- Create another array wt[] to store waiting times

of processes. Initialize this array as 0.

3- Initialize time : t = 0

4- Keep traversing the all processes while all processes

are not done. Do following for i'th process if it is

not done yet.

a- If rem\_bt[i] > quantum

(i) t = t + quantum

(ii) bt\_rem[i] -= quantum;

c- Else // Last cycle for this process

(i) t = t + bt\_rem[i];

(ii) wt[i] = t - bt[i]

(ii) bt\_rem[i] = 0; // This process is over

**Program:**

#include<iostream>

#include<queue>

using namespace std;

int main()

{

int n;

cout<<"Enter no of processes ";

cin>>n;

bool completed[n];

bool inQ[n];

int arrivalTime[n];

int burstTime[n],bT[n];

int completeTime[n]={0};

int turnAroundTime[n]={0};

int waitTime[n]={0};

cout<<"Enter arrival time ";

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

{

completed[i]=false;

inQ[i]=false;

cin>>arrivalTime[i];

}

cout<<"Enter burst time ";

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

{

cin>>burstTime[i];

bT[i]=burstTime[i];

}

int tq=2;

int currentTime=arrivalTime[0];

queue<int>Q;

Q.push(0);

inQ[0]=true;

while(!Q.empty())

{

int job=Q.front();

Q.pop();

//process the job

if(burstTime[job]>tq )

{

currentTime+=tq;

burstTime[job]-=tq;

}

else

{

currentTime+=burstTime[job];

completed[job]=true;

burstTime[job]=0;

}

if(burstTime[job]==0)

{

inQ[job]=false;

completed[job]=true;

completeTime[job]=currentTime;

turnAroundTime[job]=completeTime[job]-arrivalTime[job];

waitTime[job]=turnAroundTime[job]-bT[job];

}

bool check=false;

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

{

if(inQ[i]==false && completed[i]==false && arrivalTime[i] <= currentTime)

{

check=true;

Q.push(i);

inQ[i]=true;

}

}

if(check==false)

{

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

{

if(inQ[i]==false && completed[i]==false && arrivalTime[i] <= currentTime)

{

currentTime=arrivalTime[i];

check=true;

Q.push(i);

inQ[i]=true;

break;

}

}

}

if(burstTime[job]!=0)

{

inQ[job]=true;

Q.push(job);

}

}

cout<<"wait Time ";

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

{

cout<<waitTime[i]<<" ";

}cout<<endl;

cout<<"turn around Time ";

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

{

cout<<turnAroundTime[i]<<" ";

}cout<<endl;

cout<<"completion Time ";

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

{

cout<<completeTime[i]<<" ";

}cout<<endl;

float atat=0;

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

{

atat+=turnAroundTime[i];

}

cout<<"average turn around time: "<<atat/n<<"\n";

float awt;

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

{

awt+=waitTime[i];

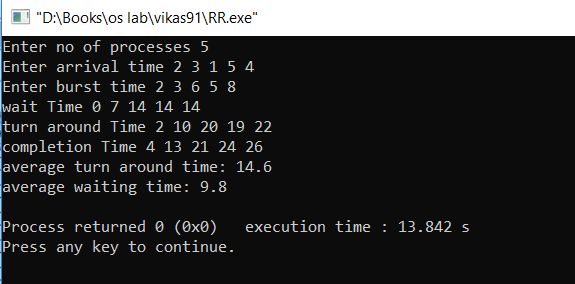
}

cout<<"average waiting time: "<<awt/5<<"\n";

return 0;

}

**Output:**



**PRACTICAL-5**

**Aim:**

Write a program for shortest remaining job first CPU scheduling.

**Theory:**

It is preemptive mode of SJF algorithm in which we give priority to the process having shortest burst time remaining.

**Algorithm:**

1. First, sort the processes in increasing order of their Arrival Time.
2. Choose the process having least arrival time but with least Burst Time. Then process it for 1 unit. Check if any other process arrives upto that time of execution or not.
3. Repeat the above both steps until execute all the processes.

**Program:**

#include <bits/stdc++.h>

using namespace std;

struct Process {

int pid;

int bt;

int art;

};

void findWaitingTime(Process proc[], int n,

int wt[])

{

int rt[n];

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

rt[i] = proc[i].bt;

int complete = 0, t = 0, minm = INT\_MAX;

int shortest = 0, finish\_time;

bool check = false;

while (complete != n) {

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

if ((proc[j].art <= t) &&

(rt[j] < minm) && rt[j] > 0) {

minm = rt[j];

shortest = j;

check = true;

}

}

if (check == false) {

t++;

continue;

}

rt[shortest]--;

minm = rt[shortest];

if (minm == 0)

minm = INT\_MAX;

if (rt[shortest] == 0) {

complete++;

check = false;

finish\_time = t + 1;

wt[shortest] = finish\_time -

proc[shortest].bt -

proc[shortest].art;

if (wt[shortest] < 0)

wt[shortest] = 0;

}

t++;

}

}

void findTurnAroundTime(Process proc[], int n,

int wt[], int tat[])

{

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

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

}

void findavgTime(Process proc[], int n)

{

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

total\_tat = 0;

findWaitingTime(proc, n, wt);

findTurnAroundTime(proc, n, wt, tat);

cout << "Processes "

<< " Burst time "

<< " Waiting time "

<< " Turn around time\n";

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

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

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

cout << " " << proc[i].pid << "\t\t"

<< proc[i].bt << "\t\t " << wt[i]

<< "\t\t " << tat[i] << endl;

}

cout << "\nAverage waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main()

{

int n;

cout<<"enter no. of processes: ";

cin>>n;

Process proc[n];

cout<<"enter arrival time and burst time for each process\n";

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

{

proc[i].pid=i+1;

cin>>proc[i].art>>proc[i].bt;

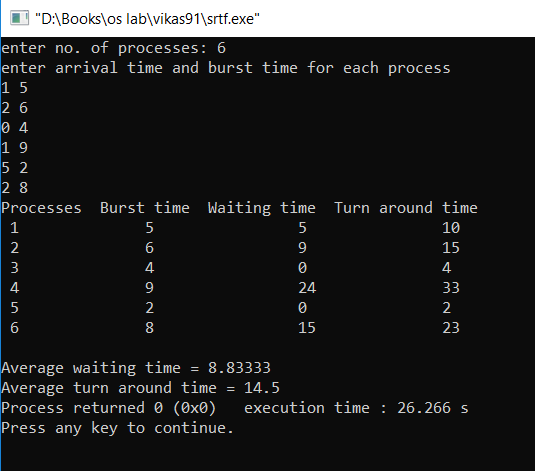
}

findavgTime(proc, n);

return 0;

}

**Output:**



**PRACTICAL 6**

**AIM**:

To implement non-preemptive priority scheduling in C++.

**THEORY :**

**Priority scheduling** is a non-preemptive **algorithm** and one of the most common **scheduling algorithms** in batch systems. Each process is assigned a **priority**. Process with highest **priority** is to be executed first and so on. Processes with same **priority** are executed on first come first served basis.

**Program**:

#include<iostream>

using namespace std;

struct process{

int pid;

int at;

int bt;

int priority;

int wt;

int tat;

int ct;

int btime;

};

int main()

{

int i,j,n;

cout<<"enter number of processes:";

cin>>n;

cout<<"enter id,priority,arrival time,burst time of the processes:\n";

cout<<"id|p|at|bt\n";

process arr[n];

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

{

cin>>arr[i].pid>>arr[i].priority>>arr[i].at>>arr[i].bt;

arr[i].btime=arr[i].bt;

}

cout<<"pid\tat\tbt\ttat\twt\tct\n";

int remain=n,time=0;

int pty,sp;

float atat,awt;

while(remain)

{

pty=-1;

sp=-1;

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

{

if((arr[i].at<=time)&&(arr[i].priority>pty)&&(arr[i].btime>0))

{

sp=i;pty=arr[i].priority;

}

}

if(sp!=-1)

{

time+=arr[sp].bt;

arr[sp].btime=0;

int tt=time-arr[sp].at;

atat+=tt;

int ww=time-arr[sp].at-arr[sp].bt;

awt+=ww;

cout<<arr[sp].pid<<"\t"<<arr[sp].at<<"\t"<<arr[sp].bt<<"\t"<<tt<<"\t"<<ww<<"\t"<<time<<"\n";

remain--;

}

else

time++;

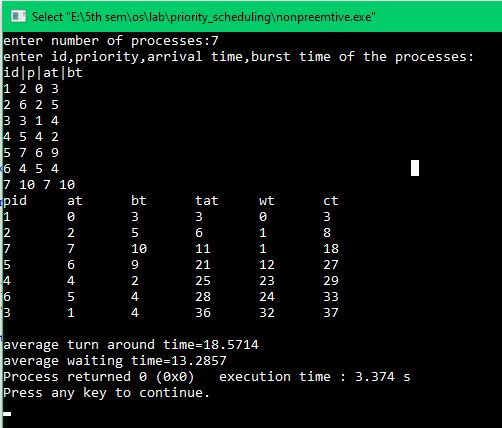
}

cout<<"\naverage turn around time="<<atat/n;

cout<<"\naverage waiting time="<<awt/n;

}

**Output:**



**PRACTICAL-7**

**AIM**:

To implement preemptive priority scheduling in C++.

**THEORY :**

**Priority scheduling** is a non-preemptive **algorithm** and one of the most common **scheduling algorithms** in batch systems. Each process is assigned a **priority**. Process with highest **priority** is to be executed first and so on. Processes with same **priority** are executed on first come first served basis.

**PROGRAM:**

#include<iostream>

using namespace std;

struct process{

int pid;

int at;

int bt;

int priority;

int wt;

int tat;

int ct;

int btime;

};

int main()

{

int i,j,n;

cout<<"enter number of processes:";

cin>>n;

cout<<"enter id,arrival time,burst time,priority of the processes:\n";

cout<<"id|at|bt|p\n";

process arr[n];

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

{

cin>>arr[i].pid>>arr[i].at>>arr[i].bt>>arr[i].priority;

arr[i].btime=arr[i].bt;

}

cout<<"pid\tat\tbt\ttat\twt\tct\n";

int remain=n,time=0;

int pty,sp;

float atat,awt;

while(remain){

pty=-1;

sp=-1;

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

{

if((arr[i].at<=time)&&(arr[i].priority>pty)&&(arr[i].btime>0))

{

sp=i;pty=arr[i].priority;

// cout<<sp<<"\n";

}

}

time++;

if(sp!=-1)

{

arr[sp].btime--;

if(arr[sp].btime==0)

{

int tt=time-arr[sp].at;

atat+=tt;

int ww=time-arr[sp].at-arr[sp].bt;

awt+=ww;

cout<<arr[sp].pid<<"\t"<<arr[sp].at<<"\t"<<arr[sp].bt<<"\t"<<tt<<"\t"<<ww<<"\t"<<time<<"\n";

remain--;

}

}

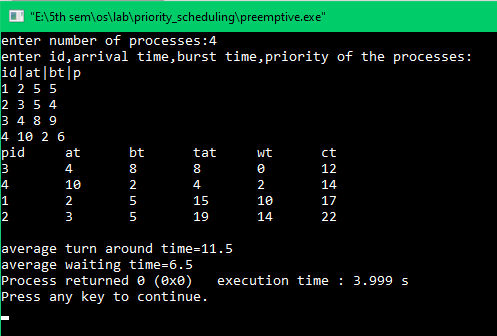
}

cout<<"\naverage turn around time="<<atat/n;

cout<<"\naverage waiting time="<<awt/n;

}

**Output:**



**PRACTICAL 8**

**AIM**:

To implement Highest response ratio next scheduling (HRRN) in C++.

**THEORY:**

This is a non-preemptive **algorithm** in which, the **scheduling** is done on the basis of an extra parameter called Response Ratio. A Response Ratio is calculated for each of the available jobs and the Job with the highest response ratio is given priority over the others. Response Ratio is calculated by the given formula.

Response ratio=Response Ratio = (W+S)/S

1. W → Waiting Time
2. S → Service Time or Burst Time

**Program:**

#include<iostream>

using namespace std;

struct process{

int pid;

int at;

int bt;

int wt;

int tat;

int ct;

int btime;

};

int main()

{

int i,j,n;

cout<<"enter number of processes:";

cin>>n;

cout<<"enter id,arrival time,burst time of the processes:\n";

cout<<"id|at|bt\n";

process arr[n];

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

{

cin>>arr[i].pid>>arr[i].at>>arr[i].bt;

arr[i].btime=arr[i].bt;

}

cout<<"pid\tat\tbt\ttat\twt\tct\n";

int remain=n,time=0;

int rr,sp;

float atat,awt;

while(remain)

{

rr=-1;

sp=-1;

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

{

int prr=(time-arr[i].at+arr[i].bt)/arr[i].bt;

if((arr[i].at<=time)&&(prr>rr)&&(arr[i].btime>0))

{

sp=i;rr=prr;

}

}

if(sp!=-1)

{

time+=arr[sp].bt;

arr[sp].btime=0;

int tt=time-arr[sp].at;

atat+=tt;

int ww=time-arr[sp].at-arr[sp].bt;

awt+=ww;

cout<<arr[sp].pid<<"\t"<<arr[sp].at<<"\t"<<arr[sp].bt<<"\t"<<tt<<"\t"<<ww<<"\t"<<time<<"\n";

remain--;

}

else

time++;

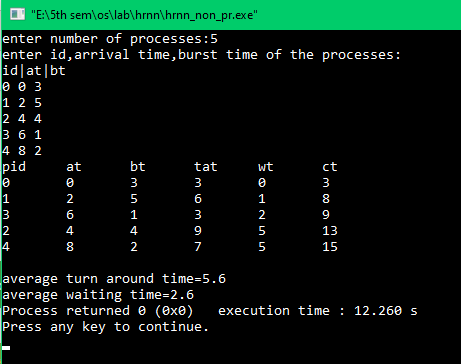
}

cout<<"\naverage turn around time="<<atat/n;

cout<<"\naverage waiting time="<<awt/n;

}

**Output:**



**PRACTICAL 9**

**AIM**:

To implement Mutual exclusion algorithms in any programming language

**THEORY :**

A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access of the resource. Upon releasing the resource, the thread unlocks the mutex.

1. **Algorithm 1:**

#include<iostream>

using namespace std;

void sync\_func(string s,int n)

{

string prev;

while(n--){

if(s=="p1")

{

while(s=="p2")

{

}

cout<<"process "<<s<<" is executing in critical section\n";

prev=s;

s="p2";

cout<<prev<<" executing other processes\n";

cout<<prev<<" exiting\n";

}

else if(s=="p2")

{

while(s=="p1")

{

}

cout<<"process "<<s<<" is executing in critical section\n";

prev=s;

s="p1";

cout<<prev<<" executing other processes\n";

cout<<prev<<" exiting\n";

}

}

}

int main()

{

string process;

int n;

cout<<"enter which process will start first (p1,p2):";

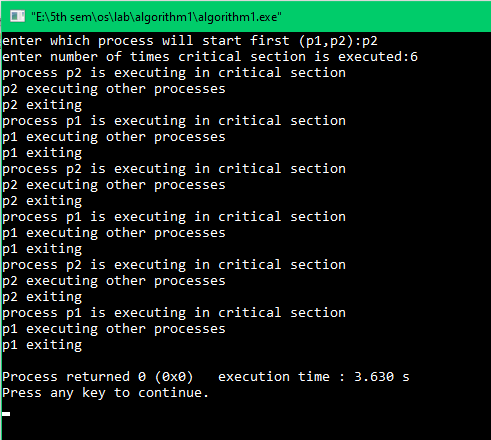
cin>>process;

cout<<"enter number of times critical section is executed:";

cin>>n;

sync\_func(process,n);}

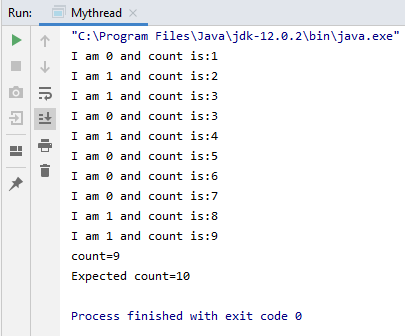
**Output:-**

****

**2) Algorithm 2:-**

**public class** Mythread **extends** Thread {  
  
 **public int thread\_id**;  
 **public static final int *countToThis***=10;  
 **public static final int *numberOfThreads***=2;  
 **public static volatile int** *count*=0;  
 **private static volatile boolean**[] *flag*=**new boolean**[***numberOfThreads***];  
  
 **public** Mythread(**int** id){  
 **thread\_id**=id;  
 }  
  
 **public void** run(){  
 **int** scale=2;  
 **for**(**int** j=0;j<***countToThis***;++j){  
 *count*++;  
 **while**(*flag*[1-**this**.**thread\_id**]){}  
 *flag*[**this**.**thread\_id**]=**true**;  
 System.***out***.println(**"I am "**+**thread\_id**+**" and count is:"**+*count*);  
 *flag*[**this**.**thread\_id**]=**false**;  
 **try**{  
 *sleep*((**int**) (Math.*random*() \* scale));  
 }  
 **catch** (InterruptedException e){  
 e.printStackTrace();  
 }  
 }  
 }  
  
  
 **public static void** main(String[] args){  
 Mythread[] mythreads=**new** Mythread[***numberOfThreads***];  
 **for**(**int** i=0;i<***numberOfThreads***;++i){  
 mythreads[i]=**new** Mythread(i);  
 mythreads[i].start();  
 }  
  
 **for**(**int** i=0;i<***numberOfThreads***;++i){  
 **try**{  
 mythreads[i].join();  
 }  
 **catch** (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 }  
  
 System.***out***.println(**"count="**+*count*);  
 System.***out***.println(**"Expected count="**+(***countToThis***\****numberOfThreads***));  
 }  
}

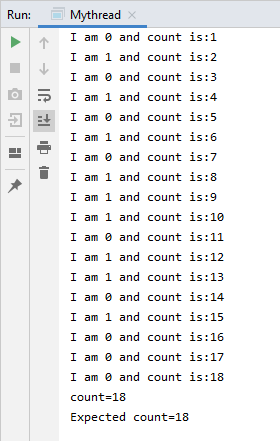
**Output:-**



**3) Algorithm 3:-**

**public class** Mythread **extends** Thread {  
  
 **public int thread\_id**;  
 **public static final int *countToThis***=10;  
 **public static final int *numberOfThreads***=2;  
 **public static volatile int** *count*=0;  
 **private static volatile boolean**[] *flag*=**new boolean**[***numberOfThreads***];  
  
 **public** Mythread(**int** id){  
 **thread\_id**=id;  
 }  
  
 **public void** run(){  
 **int** scale=2;  
 **for**(**int** j=0;j<***countToThis***;++j){  
 *count*++;  
 *flag*[**this**.**thread\_id**]=**true**;  
 **while**(*flag*[1-**this**.**thread\_id**]){}  
 System.***out***.println(**"I am "**+**thread\_id**+**" and count is:"**+*count*);  
 *flag*[**this**.**thread\_id**]=**false**;  
 **try**{  
 *sleep*((**int**) (Math.*random*() \* scale));  
 }  
 **catch** (InterruptedException e){  
 e.printStackTrace();  
 }  
 }  
 }  
  
  
 **public static void** main(String[] args){  
 Mythread[] mythreads=**new** Mythread[***numberOfThreads***];  
 **for**(**int** i=0;i<***numberOfThreads***;++i){  
 mythreads[i]=**new** Mythread(i);  
 mythreads[i].start();  
 }  
  
 **for**(**int** i=0;i<***numberOfThreads***;++i){  
 **try**{  
 mythreads[i].join();  
 }  
 **catch** (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 }  
  
 System.***out***.println(**"count="**+*count*);  
 System.***out***.println(**"Expected count="**+(***countToThis***\****numberOfThreads***));  
 }  
}

**Output: -**



**PRACTICAL** **10**

**AIM: -**

Implement Dekker’s algorithm for mutual exclusion in any programming Language

**Description:**

Dekker’s algorithm was the first provably-correct solution to the critical section problem. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

*Algorithm:-*

*var flag: array [0..1] of boolean;*

*turn: 0..1;*

*repeat*

*flag[i] := true;*

*while flag[j] do*

*if turn = j then*

*begin*

*flag[i] := false;*

*while turn = j do no-op;*

*flag[i] := true;*

*end;*

*critical section*

*turn := j;*

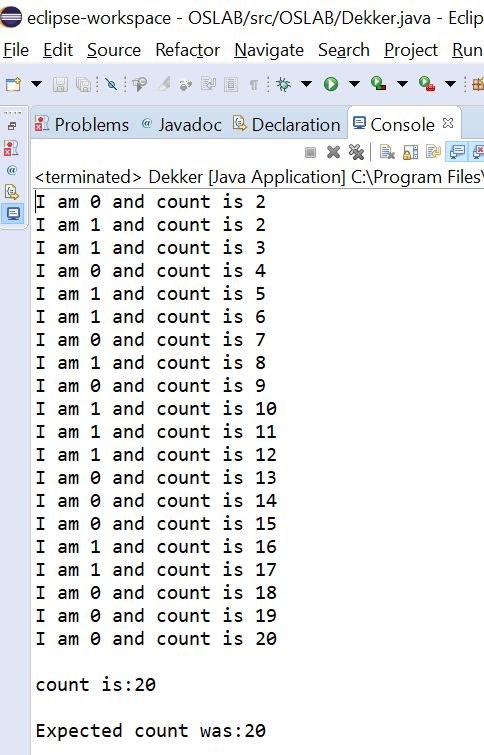
*flag[i] := false;*

*remainder section*

*until false;*

**Program: -**

**public class** Dekker **extends** Thread{  
  
 **public int thread\_id**;  
 **public static volatile int** *turn*;  
 **public static final int *countToThis***=10;  
 **public static final int *numberOfThreads***=2;  
 **public static volatile int** *count*=0;  
 **public static volatile boolean**[] *flag*=**new boolean**[***numberOfThreads***];  
  
  
 **public** Dekker(**int** id){  
 **thread\_id**=id;  
 }  
  
 **public void** run(){  
  
 **int** scale=10;  
 **for**(**int** i=0;i<***countToThis***;++i){  
 *count*++;  
 *flag*[**this**.**thread\_id**]=**true**;  
 **while**(*flag*[1-**this**.**thread\_id**]){  
 **if**(*turn*==1-**thread\_id**){  
 *flag*[**this**.**thread\_id**]=**false**;  
 **while**(*turn*==1-**thread\_id**){}  
 *flag*[**this**.**thread\_id**]=**true**;  
 }  
 }  
 System.***out***.println(**"I am "**+**thread\_id**+**" and count is "**+*count*);  
 *turn*=1-**thread\_id**;  
 *flag*[**this**.**thread\_id**]=**false**;  
 **try**{  
 *sleep*((**int**) (Math.*random*() \* scale));  
 }  
 **catch** (InterruptedException e){  
  
 }  
  
 }  
 }  
 **public static void** main(String[] args){  
 Dekker[] threads=**new** Dekker[***numberOfThreads***];  
 **for**(**int** i=0;i<threads.**length**;++i){  
 threads[i]=**new** Dekker(i);  
 threads[i].start();  
 }  
  
  
 **for**(**int** i=0;i<threads.**length**;++i){  
 **try**{  
 threads[i].join();  
 }  
 **catch** (InterruptedException e){  
 e.printStackTrace();  
 }  
 }  
  
 System.***out***.println(**"\ncount is:"**+*count*);  
 System.***out***.println(**"\nExpected count was:"**+(***countToThis***\****numberOfThreads***));  
 }  
}



**PRACTICAL 11**

**AIM:** Implement Lamport Bakery algorithm for mutual exclusion .

**Description: -**

The **Bakery algorithm** is one of the simplest known solutions to the mutual exclusion problem for the general case of N process. Bakery Algorithm is a critical section solution for **N** processes. The algorithm preserves the first come first serve property.

**ALGORITHM :**

repeat

choosing[i] := true;

number[i] := max(number[0], number[1], ..., number[n - 1])+1;

choosing[i] := false;

for j := 0 to n - 1

do begin

while choosing[j] do no-op;

while number[j] != 0

and (number[j], j) < (number[i], i) do no-op;

end;

critical section

number[i] := 0;

remainder section

until false;

**PROGRAM :**

package OSLAB;

public class Bakery extends Thread {

public int thread\_id;

public static final int countToThis = 4;

public static final int numberOfThreads = 5;

public static volatile int count = 0;

private static volatile boolean[] choosing = new boolean[numberOfThreads];

private static volatile int[] ticket = new int[numberOfThreads];

public Bakery(int id) {

thread\_id = id;

}

public void run() {

int scale = 2;

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

lock(thread\_id);

count = count + 1;

System.out.println("I am " + thread\_id + " and count is: " + count);

try {

sleep((int) (Math.random() \* scale));

} catch (InterruptedException e) { }

unlock(thread\_id);

}

}

public void lock(int id) {

choosing[id] = true;

ticket[id] = findMax() + 1;

choosing[id] = false;

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

if (j == id)

continue;

while (choosing[j]) { }

while (ticket[j] != 0 && (ticket[id] > ticket[j] || (ticket[id] == ticket[j] && id > j))) { /\* nothing \*/ }

}

}

private void unlock(int id) {

ticket[id] = 0;

}

private int findMax() {

int m = ticket[0];

for (int i = 1; i < ticket.length; i++) {

if (ticket[i] > m)

m = ticket[i];

}

return m;

}

public static void main(String[] args) {

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

choosing[i] = false;

ticket[i] = 0;

}

Bakery[] threads = new Bakery[numberOfThreads];

for (int i = 0; i < threads.length; i++) {

threads[i] = new Bakery(i);

threads[i].start();

}

for (int i = 0; i < threads.length; i++) {

try {

threads[i].join();

} catch (InterruptedException e) {

e.printStackTrace();

}

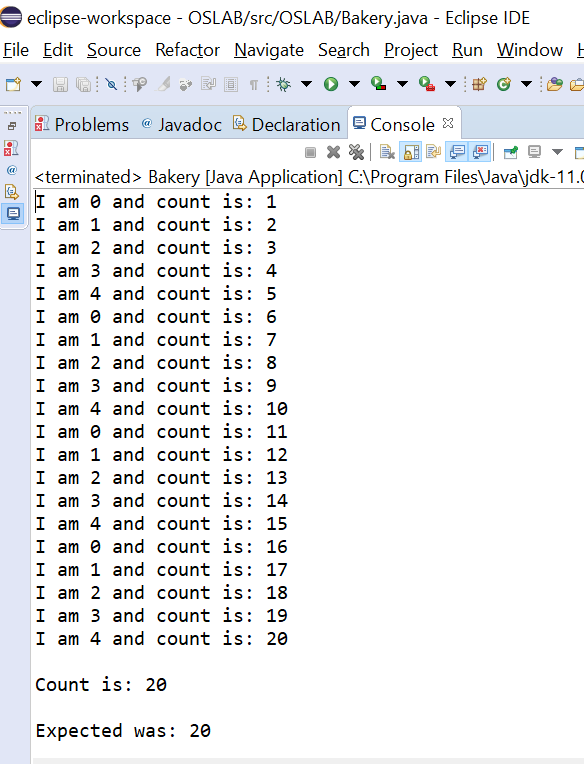
}

System.out.println("\nCount is: " + count);

System.out.println("\nExpected was: " + (countToThis \* numberOfThreads));

}

}



**PRACTICAL 12**

**AIM: -**

WAP to implement solution to producer consumer problem.

**THEORY: -**  
 We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section

.

**ALGORITHM** :

do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)

PROGRAM :

**package** OSLAB;

**import** java.util.concurrent.Semaphore;

**class** Q

{

**int** item;

**static** Semaphore *semCon* = **new** Semaphore(0);

**static** Semaphore *semProd* = **new** Semaphore(1);

**void** get()

{

**try** {

*semCon*.acquire();

}

**catch**(InterruptedException e) {

System.***out***.println("InterruptedException caught");

}

System.***out***.println("Consumer consumed item : " + item);

*semProd*.release();

}

**void** put(**int** item)

{

**try** {

*semProd*.acquire();

} **catch**(InterruptedException e) {

System.***out***.println("InterruptedException caught");

}

**this**.item = item;

System.***out***.println("Producer produced item : " + item);

*semCon*.release();

}

}

**class** Producer **implements** Runnable

{

Q q;

Producer(Q q) {

**this**.q = q;

**new** Thread(**this**, "Producer").start();

}

**public** **void** run() {

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

q.put(i);

}

}

**class** Consumer **implements** Runnable

{

Q q;

Consumer(Q q){

**this**.q = q;

**new** Thread(**this**, "Consumer").start();

}

**public** **void** run()

{

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

q.get();

}

}

**class** PC

{

**public** **static** **void** main(String args[])

{

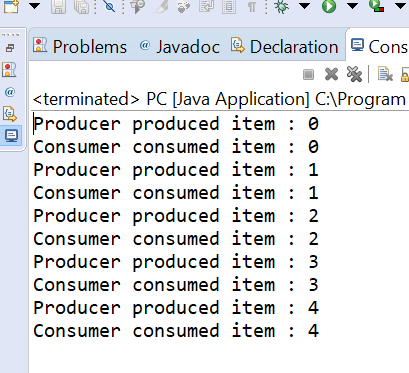
Q q = **new** Q();

**new** Consumer(q);

**new** Producer(q);

}

}



**PRACTICAL 13**

**AIM:**

WAP to implement solution to reader-writer problem.

**THEORY:**  
Consider a situation where we have a file shared between many people.

* If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
* However if some person is reading the file, then others may read it at the same time.

ALGORITHM :

WRITER PROCESS:

do {

// writer requests for critical section

wait(wrt);

// performs the write

// leaves the critical section

signal(wrt);

} while(true);

READER PROCESS:

do {

// Reader wants to enter the critical section

wait(mutex);

// The number of readers has now increased by 1

readcnt++;

// there is atleast one reader in the critical section

// this ensure no writer can enter if there is even one reader

// thus we give preference to readers here

if (readcnt==1)

wait(wrt);

// other readers can enter while this current reader is inside

// the critical section

signal(mutex);

// current reader performs reading here

wait(mutex); // a reader wants to leave

readcnt--;

// that is, no reader is left in the critical section,

if (readcnt == 0)

signal(wrt); // writers can enter

signal(mutex); // reader leaves

} while(true);

PROGRAM :

#include <iostream>

#include <pthread.h>

#include <unistd.h>

using namespace std;

class monitor {

private:

int rcnt;

int wcnt;

int waitr;

int waitw;

pthread\_cond\_t canread;

pthread\_cond\_t canwrite;

pthread\_mutex\_t condlock;

public:

monitor()

{

rcnt = 0;

wcnt = 0;

waitr = 0;

waitw = 0;

pthread\_cond\_init(&canread, NULL);

pthread\_cond\_init(&canwrite, NULL);

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

}

void beginread(int i)

{

pthread\_mutex\_lock(&condlock);

if (wcnt == 1 || waitw > 0) {

waitr++;

pthread\_cond\_wait(&canread, &condlock);

waitr--;

}

rcnt++;

cout << "reader " << i << " is reading\n";

pthread\_mutex\_unlock(&condlock);

pthread\_cond\_broadcast(&canread);

}

void endread(int i)

{

pthread\_mutex\_lock(&condlock);

if (--rcnt == 0)

pthread\_cond\_signal(&canwrite);

pthread\_mutex\_unlock(&condlock);

}

void beginwrite(int i)

{

pthread\_mutex\_lock(&condlock);

if (wcnt == 1 || rcnt > 0) {

++waitw;

pthread\_cond\_wait(&canwrite, &condlock);

--waitw;

}

wcnt = 1;

cout << "writer " << i << " is writing\n";

pthread\_mutex\_unlock(&condlock);

}

void endwrite(int i)

{

pthread\_mutex\_lock(&condlock);

wcnt = 0;

if (waitr > 0)

pthread\_cond\_signal(&canread);

else

pthread\_cond\_signal(&canwrite);

pthread\_mutex\_unlock(&condlock);

}

}

M;

void\* reader(void\* id)

{

int c = 0;

int i = \*(int\*)id;

while (c < 5) {

usleep(1);

M.beginread(i);

M.endread(i);

c++;

}

}

void\* writer(void\* id)

{

int c = 0;

int i = \*(int\*)id;

while (c < 5) {

usleep(1);

M.beginwrite(i);

M.endwrite(i);

c++;

}

}

int main()

{

pthread\_t r[5], w[5];

int id[5];

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

id[i] = i;

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

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

}

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

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

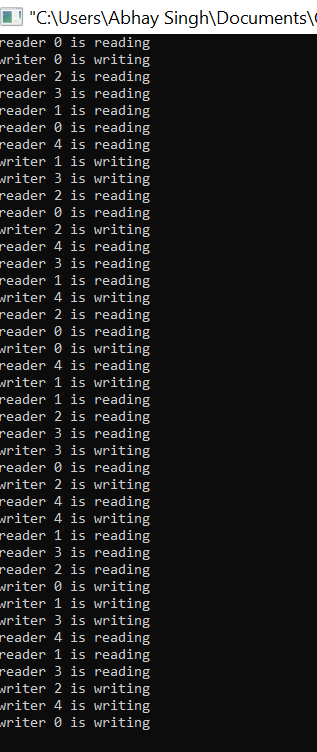
}

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

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

}

}



**PRACTICAL 14**

**AIM:**

WAP to implement solution to dining philosopher problem.

**THEORY: -**  
The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

ALGORITHM :

process P[i]

while true do

{ THINK;

PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);

EAT;

PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])

}

PROGRAM :

#include<stdio.h>

#define n 4

int compltedPhilo = 0,i;

struct fork{

int taken;

}ForkAvil[n];

struct philosp{

int left;

int right;

}Philostatus[n];

void goForDinner(int philID){ /

if(Philostatus[philID].left==10 && Philostatus[philID].right==10)

printf("Philosopher %d completed his dinner\n",philID+1);

else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){

printf("Philosopher %d completed his dinner\n",philID+1);

Philostatus[philID].left = Philostatus[philID].right = 10;

int otherFork = philID-1;

if(otherFork== -1)

otherFork=(n-1);

ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;

printf("Philosopher %d released fork %d and fork %d\n",philID+1,philID+1,otherFork+1);

compltedPhilo++;

}

else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){

if(philID==(n-1)){

if(ForkAvil[philID].taken==0){

ForkAvil[philID].taken = Philostatus[philID].right = 1;

printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);

}else{

printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);

}

}else{

int dupphilID = philID;

philID-=1;

if(philID== -1)

philID=(n-1);

if(ForkAvil[philID].taken == 0){

ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;

printf("Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1);

}else{

printf("Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1);

}

}

}

else if(Philostatus[philID].left==0){

if(philID==(n-1)){

if(ForkAvil[philID-1].taken==0){

ForkAvil[philID-1].taken = Philostatus[philID].left = 1;

printf("Fork %d taken by philosopher %d\n",philID,philID+1);

}else{

printf("Philosopher %d is waiting for fork %d\n",philID+1,philID);

}

}else{

if(ForkAvil[philID].taken == 0){

ForkAvil[philID].taken = Philostatus[philID].left = 1;

printf("Fork %d taken by Philosopher %d\n",philID+1,philID+1);

}else{

printf("Philosopher %d is waiting for Fork %d\n",philID+1,philID+1);

}

}

}else{}

}

int main(){

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

ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n){

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

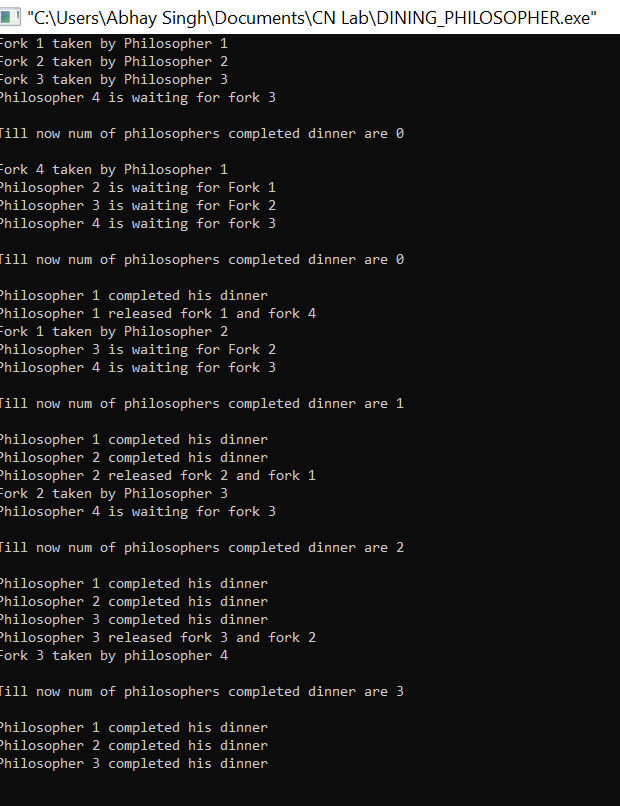
goForDinner(i);

printf("\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo);

}

return 0;

}



**PRACTICAL 15**

**AIM:**

WAP to implement solution to deadlock problem using bankers algorithm

**THEORY: -**  
The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

**ALGORITHM :**

Let **‘n’**be the number of processes in the system and **‘m’**be the number of resources types.

**Available :**

* It is a 1-d array of size **‘m’** indicating the number of available resources of each type.
* Available[ j ] = k means there are **‘k’** instances of resource type **Rj**

**Max :**

* It is a 2-d array of size ‘**n\*m’**that defines the maximum demand of each process in a system.
* Max[ i, j ] = k means process **Pi** may request at most **‘k’** instances of resource type **Rj.**

**Allocation :**

* It is a 2-d array of size**‘n\*m’**that defines the number of resources of each type currently allocated to each process.
* Allocation[ i, j ] = k means process **Pi** is currently allocated **‘k’** instances of resource type **Rj**

**Need :**

* It is a 2-d array of size **‘n\*m’** that indicates the remaining resource need of each process.
* Need [ i, j ] = k means process **Pi** currently allocated **‘k’** instances of resource type **Rj**
* Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ]

Allocationi specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.

Banker’s algorithm consist of Safety algorithm and Resource request algorithm

**Safety Algorithm**

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.  
   Initialize: Work= Available  
   Finish [i]=false; for i=1,2,……,n
2. Find an i such that both  
   a) Finish [i]=false  
   b) Need\_i<=work  
     
   if no such i exists goto step (4)
3. Work=Work + Allocation\_i  
   Finish[i]= true  
   goto step(2)
4. If Finish[i]=true for all i,  
   then the system is in safe state.

**Safe sequence is the sequence in which the processes can be safely executed.**

In this post, implementation of Safety algorithm of Banker’s Algorithm is done.

**PROGRAM :**

**package** OSLAB;

**import** java.util.\*;

**class** Bankers

{

**static** **int** *P* = 5;

**static** **int** *R* = 3;

**static** **void** calculateNeed(**int** need[][], **int** maxm[][],

**int** allot[][])

{

**for** (**int** i = 0 ; i < *P* ; i++)

**for** (**int** j = 0 ; j < *R* ; j++)

need[i][j] = maxm[i][j] - allot[i][j];

}

**static** **boolean** isSafe(**int** processes[], **int** avail[], **int** maxm[][],

**int** allot[][])

{

**int** [][]need = **new** **int**[*P*][*R*];

*calculateNeed*(need, maxm, allot);

**boolean** []finish = **new** **boolean**[*P*];

**int** []safeSeq = **new** **int**[*P*];

**int** []work = **new** **int**[*R*];

**for** (**int** i = 0; i < *R* ; i++)

work[i] = avail[i];

**int** count = 0;

**while** (count < *P*)

{

**boolean** found = **false**;

**for** (**int** p = 0; p < *P*; p++)

{

**if** (finish[p] == **false**)

{

**int** j;

**for** (j = 0; j < *R*; j++)

**if** (need[p][j] > work[j])

**break**;

**if** (j == *R*)

{

**for** (**int** k = 0 ; k < *R* ; k++)

work[k] += allot[p][k];

safeSeq[count++] = p;

finish[p] = **true**;

found = **true**;

}

}

}

**if** (found == **false**)

{

System.***out***.print("System is not in safe state");

**return** **false**;

}

}

System.***out***.print("System is in safe state.\nSafe"

+" sequence is: ");

**for** (**int** i = 0; i < *P* ; i++)

System.***out***.print(safeSeq[i] + " ");

**return** **true**;

}

**public** **static** **void** main(String[] args)

{

System.***out***.print("Enter the number of processes :");

**int** n;

Scanner scan = **new** Scanner(System.***in***);

n = scan.nextInt();

**int** processes[] = **new** **int**[n];

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

processes[i]=i;

}

System.***out***.print("Enter the number of available resourses :");

**int** m = scan.nextInt();

**int** avail[] = **new** **int**[m];

System.***out***.print("Enter the resoures available :");

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

avail[i] = scan.nextInt();

}

System.***out***.print("Enter the need matrix : ");

**int** maxm[][] = **new** **int**[n][m];

**int** allot[][] = **new** **int**[n][m];

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

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

maxm[i][j] = scan.nextInt();

}

}

System.***out***.print("Enter the allotment matrix : ");

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

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

allot[i][j] = scan.nextInt();

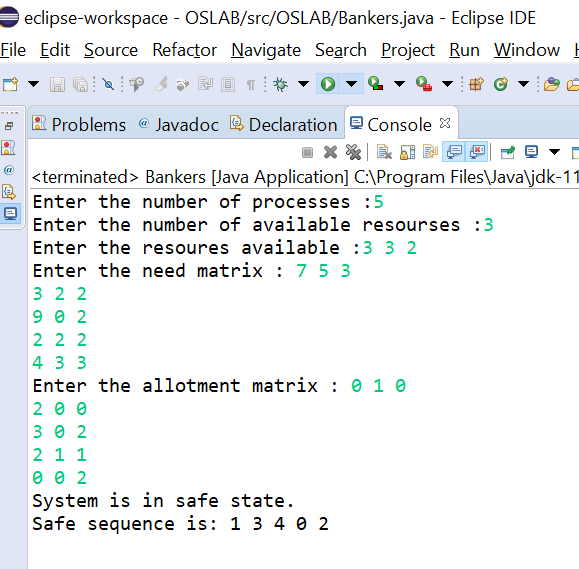
}

}

*isSafe*(processes, avail, maxm, allot);

}

}



\

**PRACTICAL 16**

**AIM: -**

Implement FIFO page replacement algorithm for Paging .

**THEORY: -**  
This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

ALGORITHM :

1- Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until

the size of set reaches capacity or all

page requests are processed.

b) Simultaneously maintain the pages in the

queue to perform FIFO.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Remove the first page from the queue

as it was the first to be entered in

the memory

b) Replace the first page in the queue with

the current page in the string.

c) Store current page in the queue.

d) Increment page faults.

1. Return page faults.

**PROGRAM :**

**package** OSLAB;

**import** java.util.HashSet;

**import** java.util.LinkedList;

**import** java.util.Queue;

**import** java.util.Scanner;

**public** **class** FIFO

{

**static** **int** pageFaults(**int** pages[], **int** n, **int** capacity)

{

HashSet<Integer> s = **new** HashSet<>(capacity);

Queue<Integer> indexes = **new** LinkedList<>() ;

**int** page\_faults = 0;

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

{

**if** (s.size() < capacity)

{

**if** (!s.contains(pages[i]))

{

s.add(pages[i]);

page\_faults++;

indexes.add(pages[i]);

}

}

**else**

{

**if** (!s.contains(pages[i]))

{

**int** val = indexes.peek();

indexes.poll();

s.remove(val);

s.add(pages[i]);

indexes.add(pages[i]);

page\_faults++;

}

}

}

**return** page\_faults;

}

**public** **static** **void** main(String args[])

{

System.***out***.print("Enter the number of pages : ");

**int** n;

Scanner scan = **new** Scanner(System.***in***);

n = scan.nextInt();

**int** pages[] = **new** **int**[n];

System.***out***.print("Enter the page numbers :");

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

**int** a = scan.nextInt();

pages[i] = a;

}

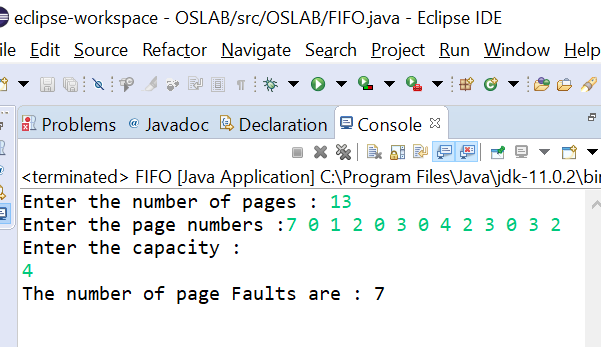
System.***out***.println("Enter the capacity : ");

**int** capacity = scan.nextInt();

System.***out***.println("The number of page Faults are : " + *pageFaults*(pages, pages.length, capacity));

}

}



**PRACTICAL 17**

**AIM: -**

Implement LRU page replacement algorithm for Paging .

**THEORY: -**

In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely.

**ALGORITHM** :

Let capacity be the number of pages that

memory can hold. Let set be the current

set of pages in memory.

1- Start traversing the pages.

i) If set holds less pages than capacity.

a) Insert page into the set one by one until

the size of set reaches capacity or all

page requests are processed.

b) Simultaneously maintain the recent occurred

index of each page in a map called indexes.

c) Increment page fault

ii) Else

If current page is present in set, do nothing.

Else

a) Find the page in the set that was least

recently used. We find it using index array.

We basically need to replace the page with

minimum index.

b) Replace the found page with current page.

c) Increment page faults.

d) Update index of current page.

1. Return page faults.

**PROGRAM : -**

**package** OSLAB;

**import** java.util.Scanner;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** LRU

{

**static** **int** pageFaults(**int** pages[], **int** n, **int** capacity)

{

HashSet<Integer> s = **new** HashSet<>(capacity);

HashMap<Integer, Integer> indexes = **new** HashMap<>();

**int** page\_faults = 0;

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

{

**if** (s.size() < capacity)

{

**if** (!s.contains(pages[i]))

{

s.add(pages[i]);

page\_faults++;

}

indexes.put(pages[i], i);

}

**else**

{

**if** (!s.contains(pages[i]))

{

**int** lru = Integer.***MAX\_VALUE***, val=Integer.***MIN\_VALUE***;

Iterator<Integer> itr = s.iterator();

**while** (itr.hasNext()) {

**int** temp = itr.next();

**if** (indexes.get(temp) < lru)

{

lru = indexes.get(temp);

val = temp;

}

}

s.remove(val);

indexes.remove(val);

s.add(pages[i]);

page\_faults++;

}

indexes.put(pages[i], i);

}

}

**return** page\_faults;

}

**public** **static** **void** main(String args[])

{

System.***out***.print("Enter the number of pages : ");

**int** n;

Scanner scan = **new** Scanner(System.***in***);

n = scan.nextInt();

**int** pages[] = **new** **int**[n];

System.***out***.print("Enter the page numbers :");

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

**int** a = scan.nextInt();

pages[i] = a;

}

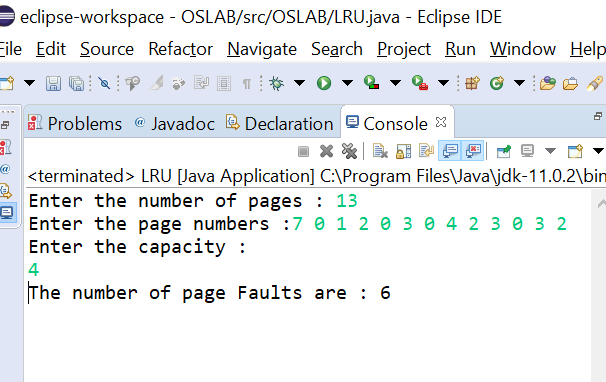
System.***out***.println("Enter the capacity : ");

**int** capacity = scan.nextInt();

System.***out***.println("The number of page Faults are : " + *1pageFaults*(pages, pages.length, capacity));

}

}



**PRACTICAL 18**

**AIM: -**

Implement OPTIMAL page replacement algorithm for Paging .

**THEORY: -**  
In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.  
In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

**ALGORITHM :**

The idea is simple, for every reference we do following :

1. If referred page is already present, increment hit count.
2. If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.

**PROGRAM :**

#include <bits/stdc++.h>

using namespace std;

bool search(int key, vector<int>& fr)

{

for (int i = 0; i < fr.size(); i++)

if (fr[i] == key)

return true;

return false;

}

int predict(int pg[], vector<int>& fr, int pn, int index)

{

int res = -1, farthest = index;

for (int i = 0; i < fr.size(); i++) {

int j;

for (j = index; j < pn; j++) {

if (fr[i] == pg[j]) {

if (j > farthest) {

farthest = j;

res = i;

}

break;

}

}

if (j == pn)

return i;

}

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

}

void optimalPage(int pg[], int pn, int fn)

{

vector<int> fr;

int hit = 0;

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

if (search(pg[i], fr)) {

hit++;

continue;

}

if (fr.size() < fn)

fr.push\_back(pg[i]);

else {

int j = predict(pg, fr, pn, i + 1);

fr[j] = pg[i];

}

}

cout << "No. of hits = " << hit << endl;

cout << "No. of misses = " << pn - hit << endl;

}

int main()

{

cout<<"Enter the number of pages : ";

int n ;

cin>>n;

int pages[n];

cout<<"Enter the pages :";

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

cin>>pages[i];

}

cout<<"Enter the capacity :";

int capacity ;

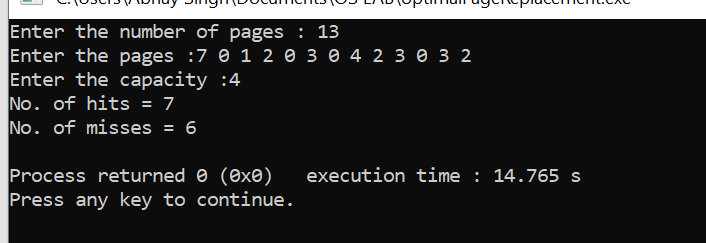
cin>>capacity;

//cout<<"The number of page faults are ";

optimalPage(pages, n, capacity);

return 0;

}



**PRACTICAL 19**

**AIM:-**

To implement SCAN Disk Schedulling in any programming language

**THEORY** : In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm.**As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

**Program: -**

#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;

void scan\_disk(int arr[],int size,int head,string dir)

{

vector<int>left,right,seek\_sequence;

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

if(arr[i]<=head){

left.push\_back(arr[i]);

}

else{

right.push\_back(arr[i]);

}

}

left.push\_back(0);

sort(left.begin(),left.end());

sort(right.begin(),right.end());

int i,j;

int seek\_count=0,curr,len;

int run=2;

while(run--){

if(dir=="left"){

for(i=left.size()-1;i>=0;--i){

curr=left[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

dir="right";

}

else{

for(i=0;i<right.size();++i){

curr=right[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

dir="left";

}

}

cout<<"seek length="<<seek\_count<<"\n";

cout<<"seek sequence=\n";

for(i=0;i<seek\_sequence.size();++i){

cout<<seek\_sequence[i]<<"\n";

}

}

int main()

{

int n,head;

cout<<"enter the size of request sequence:";

cin>>n;

int arr[n];

cout<<"enter the request sequence:";

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

cin>>arr[i];

}

cout<<"enter the head location:";

cin>>head;

string dir;

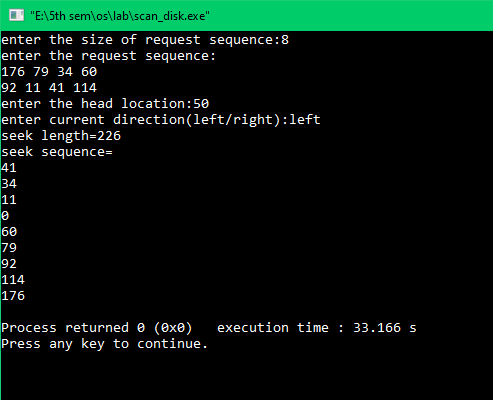
cout<<"enter current direction(left/right):";

cin>>dir;

scan\_disk(arr,n,head,dir);

}

**Output:**

****

**PRACTICAL 20**

**AIM:-**

To implement C-SCAN Disk Schedulling in any programming language

**THEORY :** Circular SCAN (C-SCAN) scheduling algorithm is a modified version of SCAN disk scheduling algorithm that deals with the inefficiency of SCAN algorithm by servicing the requests more uniformly. Like SCAN (Elevator Algorithm) C-SCAN moves the head from one end servicing all the requests to the other end. However, as soon as the head reaches the other end, it immediately returns to the beginning of the disk without servicing any requests on the return trip (see chart below) and starts servicing again once reaches the beginning. This is also known as the “Circular Elevator Algorithm” as it essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

**Program:-**

#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;

void scan\_disk(int arr[],int n,int head)

{

vector<int>left,right,seek\_sequence;

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

if(arr[i]<=head){

left.push\_back(arr[i]);

}

else{

right.push\_back(arr[i]);

}

}

sort(left.begin(),left.end());

sort(right.begin(),right.end());

int max=right[right.size()-1];

int i,j;

int seek\_count=0,curr,len;

for(i=0;i<right.size();++i){

curr=right[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

head=0;

for(i=0;i<left.size();++i){

curr=left[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

cout<<"seek length="<<seek\_count<<"\n";

cout<<"seek sequence=\n";

for(i=0;i<seek\_sequence.size();++i){

if(seek\_sequence[i]==max)

{cout<<seek\_sequence[i]<<"\n";

cout<<0<<"\n";}

else{

cout<<seek\_sequence[i]<<"\n";

}

}

}

int main()

{

int n,head;

cout<<"enter the size of request sequence:";

cin>>n;

int arr[n];

cout<<"enter the request sequence:";

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

cin>>arr[i];

}

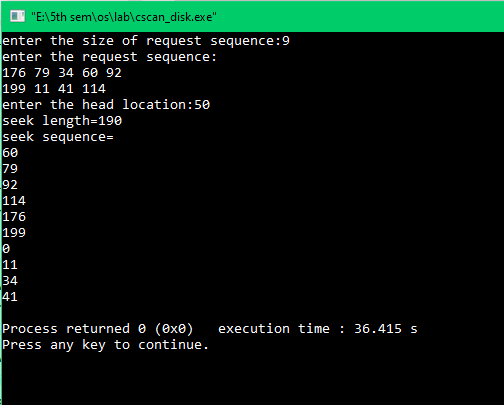
cout<<"enter the head location:";

cin>>head;

scan\_disk(arr,n,head);

}

**Output:-**



**PRACTICAL 21**

**AIM:-**

To implement LOOK Disk Schedulling in any programming language

**THEORY :**

LOOK is the advanced version of [SCAN (elevator) disk scheduling algorithm](https://www.geeksforgeeks.org/scan-elevator-disk-scheduling-algorithms/" \t "_blank) which gives slightly better seek time than any other algorithm in the hierarchy (FCFS->SRTF->SCAN->C-SCAN->LOOK). The LOOK algorithm services request similarly as SCAN algorithm meanwhile it also “looks” ahead as if there are more tracks that are needed to be serviced in the same direction. If there are no pending requests in the moving direction the head reverses the direction and start servicing requests in the opposite direction.

**Program:-**

#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;

void scan\_disk(int arr[],int size,int head,string dir)

{

vector<int>left,right,seek\_sequence;

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

if(arr[i]<=head){

left.push\_back(arr[i]);

}

else{

right.push\_back(arr[i]);

}

}

sort(left.begin(),left.end());

sort(right.begin(),right.end());

int i,j;

int seek\_count=0,curr,len;

int run=2;

while(run--){

if(dir=="left"){

for(i=left.size()-1;i>=0;--i){

curr=left[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

dir="right";

}

else{

for(i=0;i<right.size();++i){

curr=right[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

dir="left";

}

}

cout<<"seek length="<<seek\_count<<"\n";

cout<<"seek sequence=\n";

for(i=0;i<seek\_sequence.size();++i){

cout<<seek\_sequence[i]<<"\n";

}

}

int main()

{

int n,head;

cout<<"enter the size of request sequence:";

cin>>n;

int arr[n];

cout<<"enter the request sequence:";

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

cin>>arr[i];

}

cout<<"enter the head location:";

cin>>head;

string dir;

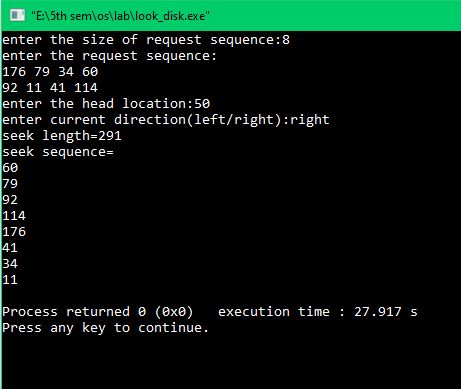
cout<<"enter current direction(left/right):";

cin>>dir;

scan\_disk(arr,n,head,dir);

}

**Output:-**



**PRACTICAL 22**

**AIM:-**

To implement C-LOOK Disk Schedulling in any programming language

**THEORY :**

**C-LOOK** is an enhanced version of both **SCAN** as well as **LOOK** disk scheduling algorithms. This algorithm also uses the idea of wrapping the tracks as a circular cylinder as C-SCAN algorithm but the seek time is better than C-SCAN algorithm. We know that C-SCAN is used to avoid starvation and services all the requests more uniformly, the same goes for C-LOOK.

**Program:-**

#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;

void scan\_disk(int arr[],int n,int head)

{

vector<int>left,right,seek\_sequence;

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

if(arr[i]<=head){

left.push\_back(arr[i]);

}

else{

right.push\_back(arr[i]);

}

}

sort(left.begin(),left.end());

sort(right.begin(),right.end());

int max=right[right.size()-1];

int i,j;

int seek\_count=0,curr,len;

for(i=0;i<right.size();++i){

curr=right[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

for(i=0;i<left.size();++i){

curr=left[i];

seek\_sequence.push\_back(curr);

len=abs(curr-head);

head=curr;

seek\_count+=len;

}

cout<<"seek length="<<seek\_count<<"\n";

cout<<"seek sequence=\n";

for(i=0;i<seek\_sequence.size();++i){

cout<<seek\_sequence[i]<<"\n";

}

}

int main()

{

int n,head;

cout<<"enter the size of request sequence:";

cin>>n;

int arr[n];

cout<<"enter the request sequence:";

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

cin>>arr[i];

}

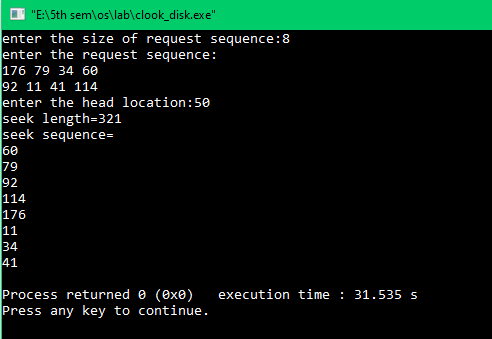
cout<<"enter the head location:";

cin>>head;

scan\_disk(arr,n,head);

}

**Output:**

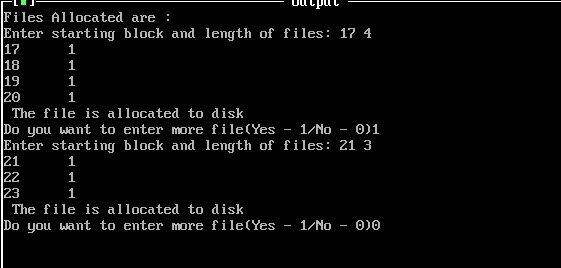
****

**PRACTICAL 23**

**AIM:** Implement Continuous File Allocation in any programming language

**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();  
}

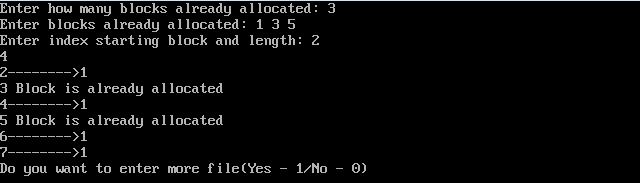


**PRACTICAL 24**

**AIM:** Implement Linked File Allocation in any programming language.

**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();  
}



**PRACTICAL 25**

**AIM:** Implement Indexed File Allocation in any programming language.

**PROGRAM:**

#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();  
}

