Experiment-1

Aim: WAP to write to a text file and then read it character wise while counting the number of alphabets, numeric characters, special characters, words, spaces and lines.

Theory: An fstream object is created and is attached to the file in out mode i.e. the write mode. The string to be written is taken as input from the user. This string is then written to the file. Then the file is opened in the in mode i.e. read mode and is read char by char till end of file is detected. And by looping the number of alphabets, numeric characters, special symbols, spaces, words are counted. The output is then displayed.

Algorithm:

Step 1: START

Step 2: fstream object created and attaches to the file in ‘out’ mode.

Step 3: Input string taken from the user.

Step 4: String written to the file.

Step 5: File closed.

Step 6: File opened in ‘in’ mode and is read character wise iteratively till eof is detected.

Step 7: Each character is tested for its ASCII value and then appropriate counters are incremented.

Step 8: Final value of all the counters is displayed.

Step 9: END

Code:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

fstream file;

file.open("test.txt",ios::out);

char input[1000];

cin.getline(input,10000);

fflush(stdin);

file<<input;

file.close();

file.open("test.txt",ios::in);

char c;

int space=0,characters=0,special=0,number=0,words=0;

while(!file.eof() && file.get(c))

{

if((c>='A' && c<='Z') || (c>='a' && c<='z'))

characters++;

else if(c==' ')

space++;

else if(c>='0' &&c<='9')

number++;

else

{

special++;

}

}

words=space+1;

cout<<"alphabets: "<<characters;

cout<<endl;

cout<<"space: "<<space;

cout<<endl;

cout<<"numbers: "<<number;

cout<<endl;

cout<<"words: "<<words;

cout<<endl;

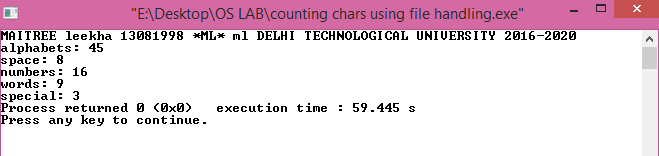
cout<<"special: "<<special;

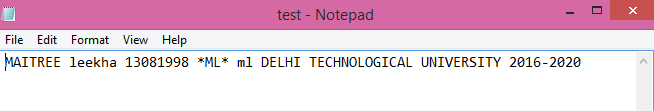
file.close();

return 0;

}

Output:





Result: We have successfully counted the required number of alphabets, numbers, special characters, spaces in the file as input by the user using file manipulation functions.

Conclusion: In this experiment we implemented fstream functions and objects and with the help of that successfully read and wrote input data from and to the file.

**Experiment-2**

Aim: To implement the First Come First Serve Algorithm on a random set of input processes from the user. Given the arrival and burst time for each process, calculate the completion time, turn around time and waiting time for each process.

Theory: Random processes are taken as input from the user with their arrival and burst time. They are pushed to a fifo queue. They are then sorted according to their arrival times in increasing order. Then after sorting, one by one the process at the front of the queue is popped and its parameters are calculated. To keep track, an extra variable, Starting Time is maintained for each process. For the first process, starting time is same as arrival time. But for the further processes, the starting time depends on the completion time of the previous process; If the completion time for the previous process is less than the arrival time of the current process then its starting time is same as its arrival time; else its starting time is the completion time of the previous process. The completion time is the time when the current process’ execution gets over. It is given by starting time plus burst time. The turn around time is calculated as the net time in CPU and is formulated as completion time minus arrival time. Waiting time is given as turn around time minus burst time.

Algorithm:

Step 1: START

Step 2: Take the process parameters- arrival times and burst times for all the processes as input from the user. Simultaneously push then to a fifo queue.

Step 3: Sort this queue in a non decreasing fashion on the basis of arrival times of the processes.

Step 4: Pop the queue while it is not empty and perform the following steps ( 5-9 ):

Here curr is the current process and prvPrcess is the previous process in the final ans vector.

Step 5: calculate the starting time:

if(size(queue)==0){curr.startTime=curr.arrivalTime;}

else if(size(queue!=0) && prevProcess.completionTime<curr.arrivalTime){

curr.startTime=curr.arrivalTime;}

else{ curr.startTime = prevProcess.completionTime;}

Step 6: completionTime = startTime+burstTime;

Step 7: TAT = completionTime-arrivalTime;

Step 8: waitingTime = TAT-burstTime;

Step 9: push the current ans parameters as a single object into the ans vector.

Step 10: print all the parameters in tabular manner.

Code:

#include<iostream>

#include<queue>

using namespace std;

class processParas

{

public:

int AT;

int BT;

};

class finalAns

{

public:

processParas p;

int ST;

int CT;

int TAT;

int WT;

};

void sortJobsAccordingToAT(queue<processParas> & processes){

vector<processParas> ans;

while(processes.size()!=0){

processParas curr= processes.front();

processes.pop();

ans.push\_back(curr);

}

for(int i=0;i<ans.size()-1;i++){

for(int j=0;j<ans.size()-i-1;j++){

if(ans[j].AT>ans[j+1].AT){

processParas temp;

temp.AT=ans[j].AT;

temp.BT=ans[j].BT;

ans[j].AT=ans[j+1].AT;

ans[j].BT=ans[j+1].BT;

ans[j+1].AT=temp.AT;

ans[j+1].BT=temp.BT;

}

}

}

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

processes.push(ans[i]);

}

}

int main()

{

queue<processParas> programQueue;

int n;

cout<<"enter the number of processes"<<'\t';

cin>>n;

cout<<endl;

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

{

processParas obj;

cout<<"enter arrival time for process "<<i+1<<" ";

cin>>obj.AT;

cout<<endl;

cout<<"enter burst time for process "<<i+1<<" ";

cin>>obj.BT;

cout<<endl;

programQueue.push(obj);

}

sortJobsAccordingToAT(programQueue);

vector<finalAns> ans;

while(programQueue.size()!=0)

{

processParas currProcess = programQueue.front();

programQueue.pop();

finalAns currAns;

currAns.p.AT=currProcess.AT;

currAns.p.BT = currProcess.BT;

if(ans.size()==0){

currAns.ST=currAns.p.AT;

}

else if(ans.size()!=0 && ans[ans.size()-1].CT<currProcess.AT)

{

currAns.ST=currAns.p.AT;

}

else if(ans.size()!=0 && ans[ans.size()-1].CT>currProcess.AT)

{

currAns.ST=ans[ans.size()-1].CT;

}

currAns.CT=currAns.ST+currAns.p.BT;

currAns.TAT=currAns.CT-currAns.p.AT;

currAns.WT= currAns.TAT-currAns.p.BT;

ans.push\_back(currAns);

}

cout<<"P\_ID"<<'\t';

cout<<"AT"<<'\t';

cout<<"BT"<<'\t';

cout<<"CT"<<'\t';

cout<<"TAT"<<'\t';

cout<<"WT"<<'\t';

cout<<endl;

for(int j=0;j<50;j++){

cout<<'-';

}

cout<<endl;

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

{

cout<<i+1<<'\t';

cout<<ans[i].p.AT<<'\t';

cout<<ans[i].p.BT<<'\t';

cout<<ans[i].CT<<'\t';

cout<<ans[i].TAT<<'\t';

cout<<ans[i].WT<<'\t';

cout<<endl;

for(int j=0;j<50;j++){

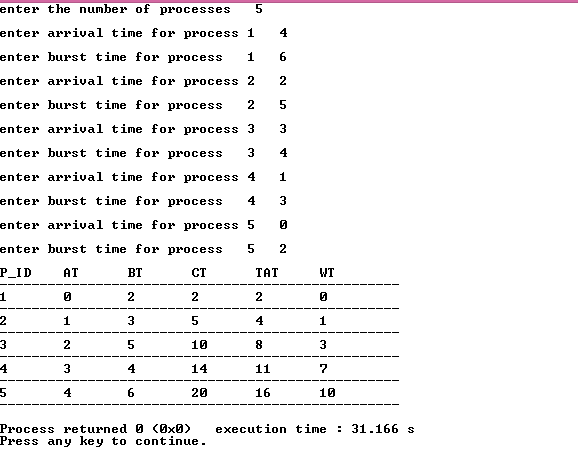
cout<<'-';

}cout<<endl;

}

}

Output:



Result: We have successfully implemented the First Come First Serve Algorithm using a fifo queue data structure. Completion time, turn around time and waiting time for each process is calculated and verified.

Conclusion: First Come First Serve algorithm grants the cpu memory to the processes as the arrive in the job queue and are executed in increasing order of arrival time i.e. the one which comes first gets executed first irrespective of the burst time.

**Experiment-3**

**Aim:** To implement the Shortest Job First Algorithm on a random set of input processes from the user. Given the burst time for each process (arrival time for all the processes is assumed to be 0), calculate the completion time, turn around time and waiting time for each process.

**Theory:** Random processes are taken as input from the user with their burst time. They are pushed to a vector of processes. This vector is sorted according to the burst times of the processes in a non decreasing order; so now the process with the least burst time is at index 0 and maximum burst time at the last index. Now iterating the vector from index 0 to last, all the parameters for the processes are calculated. For the first process, the process starts as soon as it arrives (at t=0), so its completion time is equal to the burst time plus the start time (which is zero). For the rest of the processes, the starting time is equal to prev process’ completion time and so the completion time for this process is its starting time plus its burst time. The turn around time for each process is same as that the completion time. The waiting time is given by turn around time minus the burst time.

**Algorithm:**

Step 1: START

Step 2: Take the process parameters- burst times for all the processes as input from the user. Maintain an input vector for the same.

Step 3: Sort this vector in increasing order of burst times of the processes.

Step 4: Iterate the input vector and perform the following steps for each process ( 5-9 ):

Here curr is the current process and prevPrcess is the previous process in the final ans vector. i is the iterater.

Step 5: calculate the starting time:

if(i==0){curr.startTime=0;}

else { curr.startTime=prevProcess.completionTime;}

Step 6: completionTime = startTime+burstTime;

Step 7: TAT = completionTime;

Step 8: waitingTime = TAT-burstTime;

Step 9: push the current ans parameters as a single object into the ans vector.

Step 10: print all the parameters in tabular manner.

**Code:**

#include<iostream>

#include<vector>

using namespace std;

class processParams

{

public:

int p\_id;

int BT;

};

class finalAns

{

public:

int p\_id;

int BT;

int AT=0;

int ST;

int CT;

int TAT;

int WT;

};

void sortByBurstTime(vector<processParams> & input)

{

for(int i=0; i<input.size()-1; i++)

{

for(int j=0; j<input.size()-1-i; j++)

{

if(input[j].BT>input[j+1].BT)

{

processParams temp;

temp.BT=input[j].BT;

input[j].BT= input[j+1].BT;

input[j+1].BT= temp.BT;

temp.p\_id=input[j].p\_id;

input[j].p\_id= input[j+1].p\_id;

input[j+1].p\_id= temp.p\_id;

}

}

}

}

int main()

{

int n;

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

cin>>n;

vector<processParams> input;

vector<finalAns> ans;

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

{

cout<<"Enter the burst time for process "<< i+1<<" ";

processParams tempObject;

tempObject.p\_id=i+1;

cin>>tempObject.BT;

cout<<endl;

input.push\_back(tempObject);

}

sortByBurstTime(input);

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

{

processParams currProcess=input[i];

finalAns tempAns;

tempAns.BT = currProcess.BT;

if(i==0)

{

tempAns.ST=0;

}

else

{

tempAns.ST=ans[i-1].CT;

}

tempAns.CT = tempAns.ST + tempAns.BT;

tempAns.TAT = tempAns.CT;

tempAns.WT = tempAns.TAT-tempAns.BT;

tempAns.p\_id=currProcess.p\_id;

ans.push\_back(tempAns);

}

cout<<"P\_ID"<<'\t';

cout<<"AT"<<'\t';

cout<<"BT"<<'\t';

cout<<"CT"<<'\t';

cout<<"TAT"<<'\t';

cout<<"WT"<<'\t';

cout<<endl;

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

cout<<'-';

}

cout<<endl;

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

{

cout<<ans[i].p\_id<<'\t';

cout<<ans[i].AT<<'\t';

cout<<ans[i].BT<<'\t';

cout<<ans[i].CT<<'\t';

cout<<ans[i].TAT<<'\t';

cout<<ans[i].WT<<'\t';

cout<<endl;

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

cout<<'-';

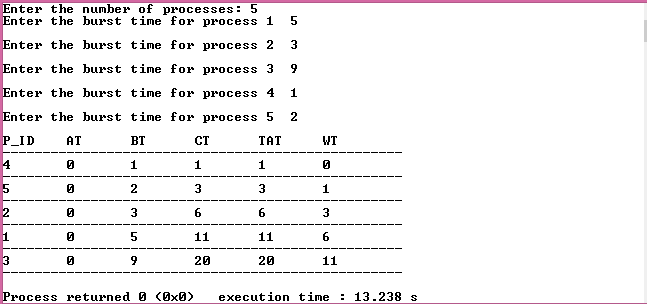
}

cout<<endl;

}

}

**Output:**



**Result:** We have successfully implemented the Shortest Job First. Completion time, turn around time and waiting time for each process is calculated and verified.

**Conclusion:** Shortest Job First algorithm grants the cpu memory to the processes in increasing order of the burst times i.e. the process with the minimum burst time is executed first and the one with the maximum burst time is executed at the last.

**Experiment-4**

**Aim:** To implement the Shortest Remaining Time First (SRTF) Algorithm on a random set of input processes from the user. Given the arrival time and the burst time for each process, calculate the completion time, turn around time and waiting time for each process.

**Theory:** Random processes are taken as input from the user with their burst time. They are pushed to a vector of processes. This vector is sorted according to the arrival times of the processes in a non decreasing order. SRTF maintains a timestamp since it has to check after every 1 unit time for the arrival of a process which has burst time less than the current process. If such a process has arrives, the current process is preempted and this new process starts executing. This check takes place after every one unit time until all the processes have been completed. We store the information regarding the execution sequence and finish time in a class object called ‘gandtChart’. Finally, all the parameters required are calculated- completion time (CT), turn around time (TAT) and waiting time (WT). From the gandtChart object, CT is same as the finish time of the corresponding process which is found using its process id (pId). The TAT is calculated as: CT-AT, i.e. the total time for which a process remains in the system. Waiting time is calculated as: TAT-BT, i.e. the time in the system other than the burst time is it’s waiting time. All the parameters for the processes are then printed in a tabular format.

**Algorithm:**

Step 1: START

Step 2: Take the process parameters- burst times for all the processes as input from the user. Maintain an input vector for the same. Also maintain a copy of this input vector (inoutCopy) which can be modified while processes are executing.

Step 3: Sort this vector in increasing order of arrival times of the processes.

Step 4: Iterate the input vector and perform the following steps until all the processes have completed their burst times. ( 5-6 ):

Here ‘curr’ is the current process and ‘t’ is the timestamp.

Step 5: curr = fetchNextProcess(input,t); It returns an object of time process. In case a process is unavailable, it returns a process with arrival and burst times as INT\_MAX which is detected if this is the case and the loop is continued and the timestamp is incremented by 1 time unit.

Step 6: If step 5 returns a valid process, its process id is compared with that of the curr process. If it is same then the finish time of the process is updated with the current time, the burst time of the process is decremented by 1 time unit, and a check is made if all the processes have been completed.

Parameters Calculated:

Step 7: For each process the following steps are followed and the ans is stored in a vector

Step 8: CT = finishTime;

Step 9: TAT = CT-AT;

Step 10: WT=TAT-BT;

Step 11: push the current ans parameters as a single object into the ans vector.

Step 12: print all the parameters in tabular manner.

**Code:**

#include<iostream>

#include<vector>

#include<climits>

using namespace std;

class process

{

public:

int pId;

int AT;

int BT;

bool complete=false;

};

class finalAns

{

public:

process p;

int ST;

int CT;

int TAT;

int WT;

bool check;

};

class gandt

{

public:

process p;

int ST;

int FT;

};

void sortByArrivalTime(vector<process> & input)

{

for(int i=0; i<input.size()-1; i++)

{

for(int j=0; j<input.size()-1-i; j++)

{

if(input[j].AT>input[j+1].AT)

{

process temp;

temp.BT=input[j].BT;

input[j].BT= input[j+1].BT;

input[j+1].BT= temp.BT;

temp.AT=input[j].AT;

input[j].AT= input[j+1].AT;

input[j+1].AT= temp.AT;

temp.pId=input[j].pId;

input[j].pId= input[j+1].pId;

input[j+1].pId= temp.pId;

}

}

}

}

void decreaseBT(vector<process> & input,int pId)

{

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

{

if(input[i].pId==pId)

{

input[i].BT=input[i].BT-1;

if(input[i].BT<=0)

{

input[i].complete=true;

}

break;

}

}

}

process fetchNextProcess(vector<process> input,int t)

{

process next;

next.pId=INT\_MAX;

next.AT=INT\_MAX;

next.BT=INT\_MAX;

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

{

if(input[i].complete==false && input[i].AT<=t && input[i].BT<next.BT && input[i].BT>0 )

{

next.pId=input[i].pId;

next.AT=input[i].AT;

next.BT=input[i].BT;

}

}

return next;

}

bool checkAllComplete(vector<process>input)

{

bool ans=true;

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

{

if(input[i].complete!=true)

{

return false;

}

}

return ans;

}

int main()

{

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

int n;

cin>>n;

cout<<endl;

vector<process> input;

vector<process>inputCopy;

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

{

process tempObject;

cout<<"Enter arrival time for process "<<i+1<<": ";

cin>>tempObject.AT;

cout<<endl;

cout<<"Enter burst time for process "<<i+1<<": ";

cin>>tempObject.BT;

cout<<endl;

tempObject.pId=i+1;

input.push\_back(tempObject);

}

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

{

process temp;

temp.AT=input[i].AT;

temp.BT=input[i].BT;

temp.pId=input[i].pId;

temp.complete=input[i].complete;

inputCopy.push\_back(temp);

}

sortByArrivalTime(input);

sortByArrivalTime(inputCopy);

int totalTime=0;

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

{

totalTime+=input[i].BT;

}

vector<gandt> gandtChart;

int t=input[0].AT;

totalTime+=t;

process currProcess= input[0];

bool allComplete=false;

while(!allComplete)

{

gandt temp;

temp.p=currProcess;

if(currProcess.BT==INT\_MAX)

{

t++;

currProcess=fetchNextProcess(input,t);

continue;

}

// cout<<"current process: "<<currProcess.pId<<endl;

temp.ST=t;

temp.FT=++t;

if(gandtChart.size()!=0 && temp.p.pId==gandtChart[gandtChart.size()-1].p.pId)

{

gandtChart[gandtChart.size()-1].FT=temp.FT;

}

else

{

gandtChart.push\_back(temp);

}

decreaseBT(input,currProcess.pId);

currProcess=fetchNextProcess(input,t);

allComplete=checkAllComplete(input);

}

/\* cout<<"pId"<<'\t'<<"ST"<<'\t'<<"FT"<<endl;

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

{

cout<<gandtChart[i].p.pId<<'\t';

cout<<gandtChart[i].ST<<'\t';

cout<<gandtChart[i].FT<<'\t';

cout<<endl;

}\*/

vector<finalAns> ans;

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

{

finalAns temp;

temp.p.AT=input[i].AT;

temp.p.BT=inputCopy[i].BT;

temp.p.pId=input[i].pId;

temp.CT=-1;

temp.TAT=-1;

temp.WT=-1;

temp.check=false;

ans.push\_back(temp);

}

int counter=0;

for(int i=gandtChart.size()-1; counter<n; i--)

{

for(int j=0; j<ans.size(); j++)

{

if(ans[j].p.pId==gandtChart[i].p.pId && ans[j].check==false)

{

ans[j].CT=gandtChart[i].FT;

ans[j].check=true;

counter++;

break;

}

}

}

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

{

ans[i].TAT=ans[i].CT-ans[i].p.AT;

ans[i].WT=ans[i].TAT-ans[i].p.BT;

}

cout<<"P\_ID"<<'\t';

cout<<"AT"<<'\t';

cout<<"BT"<<'\t';

cout<<"CT"<<'\t';

cout<<"TAT"<<'\t';

cout<<"WT"<<'\t';

cout<<endl;

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

{

cout<<'-';

}

cout<<endl;

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

{

cout<<ans[i].p.pId<<'\t';

cout<<ans[i].p.AT<<'\t';

cout<<ans[i].p.BT<<'\t';

cout<<ans[i].CT<<'\t';

cout<<ans[i].TAT<<'\t';

cout<<ans[i].WT<<'\t';

cout<<endl;

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

{

cout<<'-';

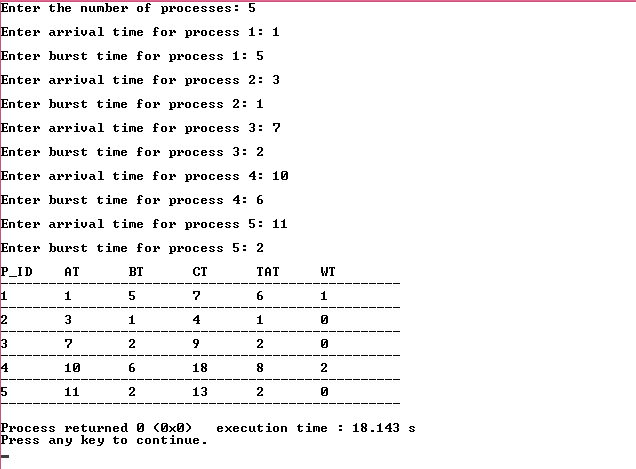
}

cout<<endl;

}

}

**Output:**



**Result:** We have successfully implemented the Shortest Remaining Time First Algorithm for CPU Scheduling. Completion time, turn around time and waiting time for each process is calculated and verified.

**Conclusion:** SRTF allots the CPU to the process which has the minimum burst time till now i.e. it progressively checks after every 1 unit time for the arrival of a shorter process and preempts the current process in that case.

**Experiment-5**

**Aim:** 5.(a) To implement Round Robin CPU scheduling algorithm on a set of processes with burst times given. (arrival time = 0)

5.(b) To implement Round Robin algorithm for a given set of processes given their burst times and arrival times as well.

**Theory:** Round Robin scheduling algorithm is preemptive by nature. It runs on the basis of time quantum. At a time a process runs no longer than 1 time quantum and after that, if the process terminates before that 1 time quantum, the remaining time is allotted to the next process, else if it takes longer than 1 time quantum, it is preempted and the algorithm now takes a circular loop over all the processes in the ready queue for at most 1 time quantum each before resuming this process again.

When the arrival times are assumed to be 0 for all the processes, the algorithm runs like a circular queue, giving at most 1 time quantum to each process and preempting it if it takes longer than 1 time quantum burst time.

When arrival time is given, the approach is a little different. At the end of each time quantum, we check for the processes that arrived during that time quantum and push them at the back of the queue before starting the next time quantum. The circular traversal is same as before.

Completion time, turn around time and waiting time for each is then calculated as: TAT= CT-AT and WT = TAT-BT.

**Algorithm:**

**5.a) Without arrival time:**

Step 1: START

Step 2: Take the process parameters- burst times for all the processes as input from the user and the value of time quantum.

Step 3: Iterate the input vector and perform the following steps until all the processes have completed their burst times. ( 5-6 ):

Step 4: Allot 1 time quantum to each process.

Step 5: If BT>1tq, it is preempted and pushed at the back of the queue and the next process is started for at most 1tq.

Step 6: If BT<1tq, it is marked finished and is popped off the ready queue and the next process is started immediately.

Parameters Calculated:

Step 7: For each process the following steps are followed and the ans is stored in a vector

Step 8: CT = finishTime;

Step 9: TAT = CT;

Step 10: WT=TAT-BT;

Step 11: push the current ans parameters as a single object into the ans vector.

Step 12: print all the parameters in tabular manner.

**5.a) With arrival time:**

Step 1: START

Step 2: Take the process parameters- burst times for all the processes as input from the user and the value of time quantum.

Step 3: Iterate the input vector and perform the following steps until all the processes have completed their burst times. ( 5-6 ):

Step 4: Allot 1 time quantum to each process.

Step 5: If BT>1tq, it is preempted and pushed at the back of the queue and the next process is started for at most 1tq.

Step 6: If BT<1tq, it is marked finished and is popped off the ready queue and the next process is started immediately.

Step 7: At the end of each time quantum, the processes that arrived in this time quantum are pushed at the back of the queue and the next time quantum then starts in the same way.

Parameters Calculated:

Step 8: For each process the following steps are followed and the ans is stored in a vector

Step 9: CT = finishTime;

Step 10: TAT = CT;

Step 11: WT=TAT-BT;

Step 12: push the current ans parameters as a single object into the ans vector.

Step 13: print all the parameters in tabular manner.

**Code:**

**5.a)**

#include<iostream>

#include<vector>

using namespace std;

//global time;

int time=0;

class processParams{

public:

int pId;

int BT;

int CT;

int TAT;

int WT;

bool check=false;

};

void execute(vector<processParams> & input,int i,int tq){

if(input[i].check==true || input[i].BT==0 ){

input[i].check=true;

return;

}

else if(input[i].BT-tq>0){

cout<<i<<"executing"<<endl;

input[i].BT-=tq;

time+=tq;

}

else if(input[i].BT-tq<0){

time+=input[i].BT;

input[i].BT=0;

input[i].CT=time;

input[i].TAT=time;

input[i].WT= input[i].TAT-input[i].BT;

input[i].check=true;

cout<<i<<"done"<<endl;

}

}

bool checkAllComplete(vector<processParams> input){

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

if(input[i].check==false){

return false;

}

}

return true;

}

int main()

{

int n;

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

cin>>n;

vector<processParams> input;

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

processParams temp;

temp.pId=i+1;

cout<<"Enter the burst time for process "<<i+1;

cin>>temp.BT;

input.push\_back(temp);

cout<<endl;

}

vector<processParams> inputcopy = input;

cout<<"Enter the time quantum ";

int tq;

cin>>tq;

while(!checkAllComplete(input)){

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

processParams curr = input[i];

execute(input,i,tq);

}

if(checkAllComplete(input)){break;}

}

cout<<"P\_ID"<<'\t';

cout<<"BT"<<'\t';

cout<<"CT"<<'\t';

cout<<"TAT"<<'\t';

cout<<"WT"<<'\t';

cout<<endl;

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

cout<<'-';

}

cout<<endl;

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

{

cout<<input[i].pId<<'\t';

cout<<inputcopy[i].BT<<'\t';

cout<<input[i].CT<<'\t';

cout<<input[i].TAT<<'\t';

cout<<input[i].WT<<'\t';

cout<<endl;

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

cout<<'-';

}

cout<<endl;

}

}

**5.b)**

#include<iostream>

#include<vector>

#include<queue>

using namespace std;

//global time;

int time=0;

class processParams

{

public:

int pId;

int AT;

int BT;

int CT;

int TAT;

int WT;

bool check=false;

};

sortBtArrivalTime(vector<processParams> & input)

{

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

{

for(int j=0; j<input.size()-i-1; j++)

{

if(input[j].AT>input[j+1].AT)

{

processParams temp;

temp.AT=input[j].AT;

input[j].AT = input[j+1].AT;

input[j+1].AT = temp.AT;

temp.BT=input[j].BT;

input[j].BT = input[j+1].BT;

input[j+1].BT = temp.BT;

temp.pId=input[j].pId;

input[j].pId = input[j+1].pId;

input[j+1].pId = temp.pId;

}

}

}

}

bool executeCurrProcess(processParams & curr, vector<processParams> & input,int tq)

{

bool flag;

cout<<"executing: "<<curr.pId<<endl;

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

{

if(input[i].pId==curr.pId)

{

if(input[i].BT==0){

return true;

}

if(input[i].BT-tq>0)

{

input[i].BT-=tq;

curr.BT-=tq;

time+=tq;

flag=false;//incomplete and push again;

}

else if(input[i].BT-tq<=0)

{

time+=input[i].BT;

input[i].BT=0;

curr.BT=0;

input[i].check=true;

curr.check=true;

input[i].CT=time;

input[i].TAT= input[i].CT-input[i].AT;

//input[i].WT = input[i].TAT- input[i].BT;

flag=true;//complete and do not push again

}

break;

}

}

return flag;

}

void pushArrivedProcesses(vector<processParams> input, queue<processParams> & rrobin, int prevTime)

{

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

{

if(input[i].AT>prevTime && input[i].AT<=time)

{

rrobin.push(input[i]);

}

}

return;

}

void checkForNewProcess(vector<processParams> input,queue<processParams> & rrobin)

{

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

{

if(input[i].AT==time && input[i].check==false)

{

rrobin.push(input[i]);

}

}

}

bool checkAllComplete(vector<processParams> input)

{

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

{

if(input[i].check==false)

{

return false;

}

}

return true;

}

int main()

{

int n;

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

cin>>n;

vector<processParams> input;

vector<processParams> inpcpy;

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

{

processParams temp;

temp.pId=i+1;

cout<<"Enter the arrival time for process "<<i+1;

cin>>temp.AT;

cout<<endl;

cout<<"Enter the burst time for process "<<i+1;

cin>>temp.BT;

input.push\_back(temp);

inpcpy.push\_back(temp);

cout<<endl;

}

cout<<"Enter the time quantum ";

int tq;

cin>>tq;

// sortByArrivalTime(input);

int startTime = input[0].AT;

queue<processParams> rrobin;

rrobin.push(input[0]);

int prevTime;

while(!checkAllComplete(input))

{

prevTime = time;

if(rrobin.empty())

{

time++;

checkForNewProcess(input,rrobin);

continue;

}

processParams curr = rrobin.front();

rrobin.pop();

bool check = executeCurrProcess(curr,input,tq);

pushArrivedProcesses(input,rrobin,prevTime);

if(check==false)

{

rrobin.push(curr);

}

}

cout<<"P\_ID"<<'\t';

cout<<"AT"<<'\t';

cout<<"BT"<<'\t';

cout<<"CT"<<'\t';

cout<<"TAT"<<'\t';

cout<<"WT"<<'\t';

cout<<endl;

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

{

cout<<'-';

}

cout<<endl;

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

{

cout<<input[i].pId<<'\t';

cout<<input[i].AT<<'\t';

cout<<inpcpy[i].BT<<'\t';

cout<<input[i].CT<<'\t';

cout<<input[i].TAT<<'\t';

cout<<input[i].TAT-inpcpy[i].BT<<'\t';

cout<<endl;

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

{

cout<<'-';

}

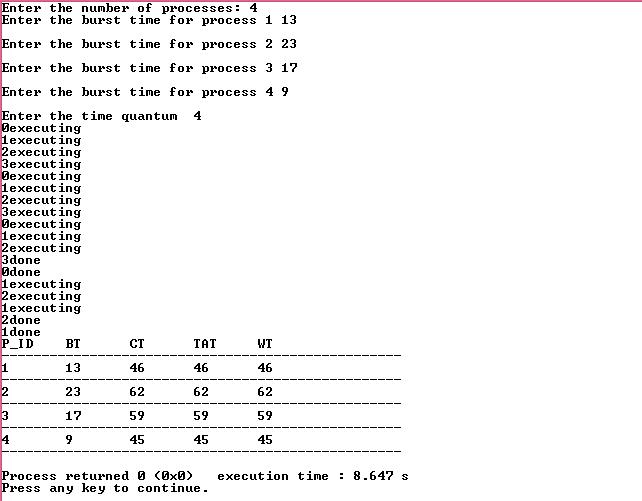
cout<<endl;

}

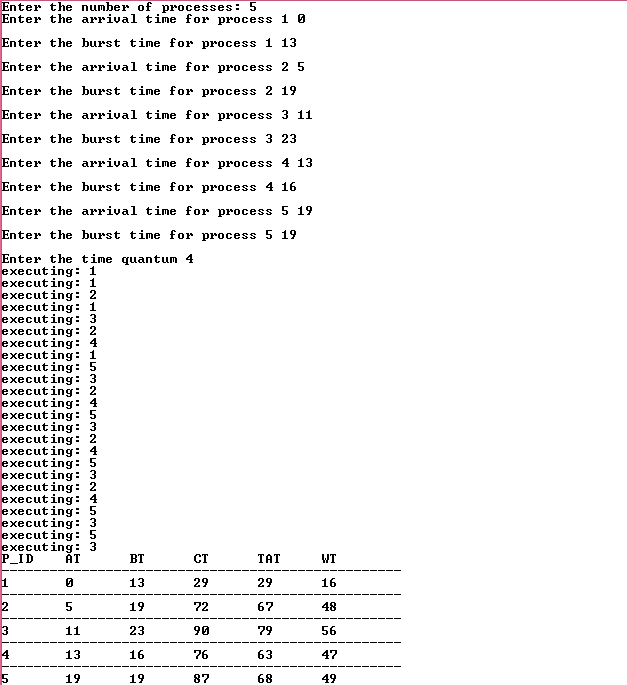
}

**Output:**

**5.a) Without Arrival Time:**

****

**5.b) With Arrival Time**

****

**Result:** We have successfully implemented the Round Robin algorithm for CPU scheduling by allotting at most 1 time quantum to each process and preempting it if it takes longer than that. The completion time, turnaround time and waiting time was calculated for both the cases; with arrival time and without arrival time.

**Conclusion:** Round Robin algorithm allots at most 1 time quantum to each process and thus, is very efficient for time sharing system where each user should not get more than his fair share of the cpu time.

**Experiment-6**

**Aim:** To implement preemptive priority scheduling algorithm.

**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 the highest priority is to be executed first and so on. Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

The scheduling starts at time t0 with the first process available. Every time a process is executed, the time stamp is incremented and the availability of a higher priority process is checked. If such a process is available, the current process is preempted and the higher priority process starts executing now. This is continued till all the processes are executed.

**Algorithm:**

Step 1: START

Step 2: Take the process parameters- burst times for all the processes as input from the user and the value of time quantum. Initialize the system clock from t = 0.

Step 3: Iterate the input vector and perform the following steps until all the processes have completed their burst times. ( 5-6 ):

Step 4: Execute the first process available and increment the time stamp by 1.

Step 5: Now check for a higher priority process. If such a process is found, preempt the current process.

Parameters Calculated:

Step 6: For each process the following steps are followed and the ans is stored in a vector

Step 7: CT = finishTime;

Step 8: TAT = CT;

Step 9: WT=TAT-BT;

Step 10: push the current ans parameters as a single object into the ans vector.

Step 11: print all the parameters in tabular manner.

**Code:**

#include<iostream>

#include<vector>

using namespace std;

//global time;

int time=0;

class processParams{

public:

int pId;

int AT;

int BT;

int priority; // lower the priority number higher the priority

int CT;

int TAT;

int WT;

bool check=false;

};

processParams fetchProcess(vector<processParams> input){

//min priority number wala process with at<=time

int i=0;

processParams p;

vector<processParams> candidates;

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

if(input[i].AT<=time && input[i].check==false){

candidates.push\_back(input[i]);

}

}

if(candidates.size()==0){

processParams temp;

temp.pId=-1;

return temp;

}

p=candidates[0];

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

if(candidates[i].priority<p.priority){

p=candidates[i];

}

}

return p;

}

void execute(vector<processParams> & input, processParams curr){

if(curr.check==true){

return;

}

cout<<"execute "<<curr.pId<<endl;

int i;

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

if(input[i].pId== curr.pId){

break;

}

}

input[i].BT-=1;

time++;

if(input[i].BT==0){

input[i].check=true;

input[i].CT=time;

}

}

bool checkAllComplete(vector<processParams> input){

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

if(input[i].check==false){

return false;

}

}

return true;

}

int main(){

int n;

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

cin>>n;

cout<<endl;

vector<processParams> input;

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

processParams temp;

temp.pId=i+1;

cout<<"Enter the arrival time for process "<<i+1<<':';

cin>>temp.AT;

cout<<endl;

cout<<"Enter the burst time for process "<<i+1<<':';

cin>>temp.BT;

cout<<endl;

cout<<"Enter the priority number for process "<<i+1<<':';

cin>>temp.priority;

cout<<endl;

input.push\_back(temp);

}

vector<processParams> inpcpy = input;

//onnput sorted by AT.

while(time<input[0].AT){

time++;

}

processParams curr;

while(!checkAllComplete(input)){

curr = fetchProcess(input);

if(curr.pId!=-1){

execute(input,curr);

}

}

cout<<endl;

cout<<"pID"<<'\t';

cout<<"AT"<<'\t';

cout<<"BT"<<'\t';

cout<<"pri"<<'\t';

cout<<"CT"<<'\t';

cout<<"TAT"<<'\t';

cout<<"WT"<<'\t';

cout<<endl;

for(int j=0; j<60; j++)

{

cout<<"-";

}

cout<<endl;

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

cout<<input[i].pId<<'\t';

cout<<input[i].AT<<'\t';

cout<<inpcpy[i].BT<<'\t';

cout<<input[i].priority<<'\t';

cout<<input[i].CT<<'\t';

cout<<input[i].CT-input[i].AT<<'\t';

cout<<input[i].CT-input[i].AT-inpcpy[i].BT<<endl;

}

for(int j=0; j<60; j++)

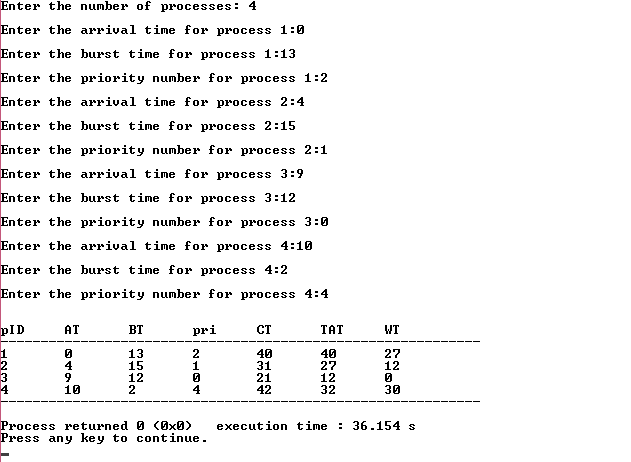
{

cout<<"-";

}

cout<<endl;

}

**Output: **

**Result:** In this experiment we successfully implemented the Pre-emptive Priority CPU Scheduling algorithm for a number of burst, arrival times and priorities.

**Conclusion:** In this experiment we learned more about the Pre-emptive Priority Scheduling algorithm and how it functions by giving priority on different criteria to a process that may be more important in the real world scenario.