**Round robin:**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62** | **#include<stdio.h>**  **int main()**  **{**  **int i, limit, total = 0, x, counter = 0, time\_quantum;**  **int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10];**  **float average\_wait\_time, average\_turnaround\_time;**  **printf("\nEnter Total Number of Processes:\t");**  **scanf("%d", &limit);**  **x = limit;**  **for(i = 0; i < limit; i++)**  **{**  **printf("\nEnter Details of Process[%d]\n", i + 1);**  **printf("Arrival Time:\t");**  **scanf("%d", &arrival\_time[i]);**  **printf("Burst Time:\t");**  **scanf("%d", &burst\_time[i]);**  **temp[i] = burst\_time[i];**  **}**  **printf("\nEnter Time Quantum:\t");**  **scanf("%d", &time\_quantum);**  **printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");**  **for(total = 0, i = 0; x != 0;)**  **{**  **if(temp[i] <= time\_quantum && temp[i] > 0)**  **{**  **total = total + temp[i];**  **temp[i] = 0;**  **counter = 1;**  **}**  **else if(temp[i] > 0)**  **{**  **temp[i] = temp[i] - time\_quantum;**  **total = total + time\_quantum;**  **}**  **if(temp[i] == 0 && counter == 1)**  **{**  **x--;**  **printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);**  **wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i];**  **turnaround\_time = turnaround\_time + total - arrival\_time[i];**  **counter = 0;**  **}**  **if(i == limit - 1)**  **{**  **i = 0;**  **}**  **else if(arrival\_time[i + 1] <= total)**  **{**  **i++;**  **}**  **else**  **{**  **i = 0;**  **}**  **}**  **average\_wait\_time = wait\_time \* 1.0 / limit;**  **average\_turnaround\_time = turnaround\_time \* 1.0 / limit;**  **printf("\n\nAverage Waiting Time:\t%f", average\_wait\_time);**  **printf("\nAvg Turnaround Time:\t%f\n", average\_turnaround\_time);**  **return 0;**  **}** |

**Pre-emptive Priority Scheduling Algorithm**

**#include<stdio.h>**

**struct process**

**{**

**char process\_name;**

**int arrival\_time, burst\_time, ct, waiting\_time, turnaround\_time, priority;**

**int status;**

**}process\_queue[10];**

**int limit;**

**void Arrival\_Time\_Sorting()**

**{**

**struct process temp;**

**int i, j;**

**for(i = 0; i < limit - 1; i++)**

**{**

**for(j = i + 1; j < limit; j++)**

**{**

**if(process\_queue[i].arrival\_time > process\_queue[j].arrival\_time)**

**{**

**temp = process\_queue[i];**

**process\_queue[i] = process\_queue[j];**

**process\_queue[j] = temp;**

**}**

**}**

**}**

**}**

**void main()**

**{**

**int i, time = 0, burst\_time = 0, largest;**

**char c;**

**float wait\_time = 0, turnaround\_time = 0, average\_waiting\_time, average\_turnaround\_time;**

**printf("\nEnter Total Number of Processes:\t");**

**scanf("%d", &limit);**

**for(i = 0, c = 'A'; i < limit; i++, c++)**

**{**

**process\_queue[i].process\_name = c;**

**printf("\nEnter Details For Process[%C]:\n", process\_queue[i].process\_name);**

**printf("Enter Arrival Time:\t");**

**scanf("%d", &process\_queue[i].arrival\_time );**

**printf("Enter Burst Time:\t");**

**scanf("%d", &process\_queue[i].burst\_time);**

**printf("Enter Priority:\t");**

**scanf("%d", &process\_queue[i].priority);**

**process\_queue[i].status = 0;**

**burst\_time = burst\_time + process\_queue[i].burst\_time;**

**}**

**Arrival\_Time\_Sorting();**

**process\_queue[9].priority = -9999;**

**printf("\nProcess Name\tArrival Time\tBurst Time\tPriority\tWaiting Time");**

**for(time = process\_queue[0].arrival\_time; time < burst\_time;)**

**{**

**largest = 9;**

**for(i = 0; i < limit; i++)**

**{**

**if(process\_queue[i].arrival\_time <= time && process\_queue[i].status != 1 && process\_queue[i].priority > process\_queue[largest].priority)**

**{**

**largest = i;**

**}**

**}**

**time = time + process\_queue[largest].burst\_time;**

**process\_queue[largest].ct = time;**

**process\_queue[largest].waiting\_time = process\_queue[largest].ct - process\_queue[largest].arrival\_time - process\_queue[largest].burst\_time;**

**process\_queue[largest].turnaround\_time = process\_queue[largest].ct - process\_queue[largest].arrival\_time;**

**process\_queue[largest].status = 1;**

**wait\_time = wait\_time + process\_queue[largest].waiting\_time;**

**turnaround\_time = turnaround\_time + process\_queue[largest].turnaround\_time;**

**printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d", process\_queue[largest].process\_name, process\_queue[largest].arrival\_time, process\_queue[largest].burst\_time, process\_queue[largest].priority, process\_queue[largest].waiting\_time);**

**}**

**average\_waiting\_time = wait\_time / limit;**

**average\_turnaround\_time = turnaround\_time / limit;**

**printf("\n\nAverage waiting time:\t%f\n", average\_waiting\_time);**

**printf("Average Turnaround Time:\t%f\n", average\_turnaround\_time);**

**}**

1. **MAIN CODE**:

#include <bits/stdc++.h>

using namespace std;

struct Process\_Data

{

int Num;

int Pid; //Process Id

int A\_time; //Process Arrival Time

int B\_time; //Process Bruest Time

int Priority; //Process Priority

int F\_time; //Process Finish Time

int R\_time; //Process Remaining Time During Execution

int W\_time; //Waiting Time

int S\_time; //Process start Time

int Res\_time;

};

struct Process\_Data current;

typedef struct Process\_Data P\_d ;

bool idsort(const P\_d& x , const P\_d& y)

{

return x.Pid < y.Pid;

}

/\*\* Sorting on the base of arrival time if that match then on Priority of Priority also match than on the base of Process Id\*\*/

bool arrivalsort( const P\_d& x ,const P\_d& y)

{

if(x.A\_time < y.A\_time)

return true;

else if(x.A\_time > y.A\_time)

return false;

if(x.Priority < y.Priority)

return true;

else if(x.Priority > y.Priority)

return false;

if(x.Pid < y.Pid)

return true;

return false;

}

bool Numsort( const P\_d& x ,const P\_d& y)

{

return x.Num < y.Num;

}

/\*Sorting on the base of Priority if that same then on the base of PID\*/

struct comPare

{

bool operator()(const P\_d& x ,const P\_d& y)

{

if( x.Priority > y.Priority )

return true;

else if( x.Priority < y.Priority )

return false;

if( x.Pid > y.Pid )

return true;

return false;

}

};

/\*\*To check the Input \*\*/

void my\_check(vector<P\_d> mv)

{

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

{

cout<<" Pid :"<<mv[i].Pid<<" \_time : "<<mv[i].A\_time<<" B\_time : "<<mv[i].B\_time<<" Priority : "<<mv[i].Priority<<endl;

}

}

int main()

{

int i;

vector< P\_d > input;

vector<P\_d> input\_copy;

P\_d temp;

int pq\_process = 0; // for PQ process

int rq\_process = 0; // for RQ process

int A\_time;

int B\_time;

int Pid;

int Priority;

int n;

int clock;

int total\_exection\_time = 0;

cin>>n;

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

{

cin>>Pid>>A\_time>>B\_time>>Priority;

temp.Num = i+1;

temp.A\_time = A\_time;

temp.B\_time = B\_time;

temp.R\_time = B\_time;

temp.Pid = Pid;

temp.Priority = Priority;

input.push\_back(temp);

}

input\_copy = input;

sort( input.begin(), input.end(), arrivalsort );

//cout<<"arrivalsort : "<<endl;

//my\_check( input ); // To check the sort unomment it

total\_exection\_time = total\_exection\_time + input[0].A\_time;

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

{

if( total\_exection\_time >= input[i].A\_time )

{

total\_exection\_time = total\_exection\_time +input[i].B\_time;

}

else

{

int diff = (input[i].A\_time - total\_exection\_time);

total\_exection\_time = total\_exection\_time + diff + B\_time;

}

}

int Ghant[total\_exection\_time]={0}; //Ghant Chart

for( i= 0; i< total\_exection\_time; i++ )

{

Ghant[i]=-1;

}

//cout<<"total\_exection\_time : "<<total\_exection\_time<<endl;

priority\_queue < P\_d ,vector<Process\_Data> ,comPare> pq; //Priority Queue PQ

queue< P\_d > rq; //Round Robin Queue RQ

int cpu\_state = 0; //idle if 0 then Idle if 1 the Busy

int quantum = 4 ; //Time Quantum

current.Pid = -2;

current.Priority = 999999;

for ( clock = 0; clock< total\_exection\_time; clock++ )

{

/\*\*Insert the process with same Arrival time in Priority Queue\*\*/

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

{

if(clock == input[j].A\_time)

{

pq.push(input[j]);

}

}

if(cpu\_state == 0) //If CPU idle

{

if(!pq.empty())

{

current = pq.top();

cpu\_state = 1;

pq\_process = 1;

pq.pop();

quantum = 4;

}

else if(!rq.empty())

{

current = rq.front();

cpu\_state = 1;

rq\_process = 1;

rq.pop();

quantum = 4;

}

}

else if(cpu\_state == 1) //If cpu has any procss

{

if(pq\_process == 1 && (!pq.empty()))

{

if(pq.top().Priority < current.Priority ) //If new process has high priority

{

rq.push(current); //push current in RQ

current = pq.top();

pq.pop();

quantum = 4;

}

}

else if(rq\_process == 1 && (!pq.empty())) //If process is from RQ and new process come in PQ

{

rq.push(current);

current = pq.top();

pq.pop();

rq\_process = 0;

pq\_process = 1;

quantum = 4 ;

}

}

if(current.Pid != -2) // Process Execution

{

current.R\_time--;

quantum--;

Ghant[clock] = current.Pid;

if(current.R\_time == 0) //If process Finish

{

cpu\_state = 0 ;

quantum = 4 ;

current.Pid = -2;

current.Priority = 999999;

rq\_process = 0;

pq\_process = 0;

}

else if(quantum == 0 ) //If time Qunatum of a current running process Finish

{

rq.push(current);

current.Pid = -2;

current.Priority = 999999;

rq\_process = 0;

pq\_process = 0;

cpu\_state=0;

}

}

}

sort( input.begin(), input.end(), idsort );

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

{

for(int k=total\_exection\_time;k>=0;k--)

{

if(Ghant[k]==i+1)

{

input[i].F\_time=k+1;

break;

}

}

}

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

{

for(int k=0;k<total\_exection\_time;k++)

{

if(Ghant[k]==i+1)

{

input[i].S\_time=k;

break;

}

}

}

sort( input.begin(), input.end(), Numsort );

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

{

input[i].Res\_time=input[i].S\_time-input[i].A\_time;

input[i].W\_time=(input[i].F\_time-input[i].A\_time)-input[i].B\_time;

}

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

{

cout<<input[i].Pid<<" "<<input[i].Res\_time<<" "<<input[i].F\_time<<" "<<input[i].W\_time<<endl;

} return 0;}