NOVEL ROUND ROBIN SCHEDULING ALGORITHM

Submitted To- PROFESSOR SANJAY PATIDAR OF DELHI TECHNOLOGICAL UNIVERSITY

Submitted by- R.Prashant (2K18/EC/124), Raghav(2K18/EC/127)

Abstract:Round Robin Scheduling is a scheduling algorithm which has been around for a long time now.In multitasking operating system, processes don't run simultaneously, but keep switching based on cpu request and I/O request. The job of the scheduler is to select a process from the ready queue and place it into the memory for execution. For this scheduling algorithms are used such as First Come First Serve (FCFS), Shortest Job First (SJF), Priority Scheduling, Round Robin (RR) Scheduling, Multilevel Queue Scheduling (MLQ). This paper proposes a new strategy of modifying the Round Robin Scheduling algorithm by making changes in its time quanta to reduce the time a process spends in waiting state. The proposed algorithm also reduces the number of context switches to provide a fair, efficient and methodical scheduling algorithm. It is an extension of Round Robin Algorithm where each of the processes is given a priority level (low, medium or high) and based on the priority level.the Time Quantum for that process is decided and executed.

Introduction:

In today's world speed is something that takes the front seat in almost all applications.Even on a Google Search the first three search results catch our attention ;also known as the golden triangle.This work has been largely inspired by this concept that the speed of scheduling must improve .To do so the Processes which arrive early and also take the most time to execute must be executed first.This can be seen in gaming as well where we require the starting of the game and loading to be done efficiently and fast so that the future gameplay is not hampered.In such a scenario our work gives importance the process which arrives first and has the highest burst time.In today's world we cannot expect our processes to run at a fixed time quantum.Hence a Dynamic Time Quantum approach has been suggested.Finally a concept of threshold time has been introduced which is basically below which a process will be executed completely if it is next in line in the waiting queue.This helps us completely eliminate the possibility of using a static time quantum and hence improves the context switching capabilities as well as the response time of the process which has come first.

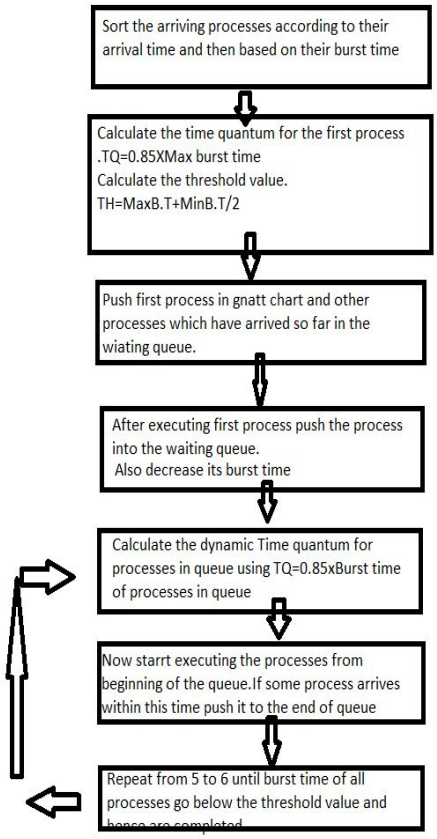
Related Work:

Your Contribution or Advantage:

Our work has been largely inspired from the work suggested in the research papers mentioned in references where a dynamic time quantum calculation is used.Our work includes improvements in the previous existing models by some amounts and is a vast improvement of the original Round Robin Scheduling Algorithm.The Round Robin Scheduling Algorithm uses a fixed value of time quantum which increases the context switching.Our aim has been to reduce the context switching.We take idea from the papers referenced and improve upon them using different techniques which include sorting the processes first based on arrival time then burst time;then using a dynamic time quantum by calculating the average of remaining burst time in the waiting queue and then using a threshold value to further preempt the working of processes.

In a brief the idea suggested in this paper is a minor improvement over the previous mentioned round robin scheduling techniques and a significant improvement over the original Round Robin Scheduling in terms of context switching.

Block Diagram of system:



Code :

The code has been attached within this github repository.
https://github.com/Prashant43226/Novel-Round-Robin-Scheduling-Algorithm

Also the code has been attached below for any future reference

Proposed work with flow chart:

1.Sort the processes according to their arrival time and then based on their burst time.
2.Calculate the time quantum for the first process.This is given by
TQ=0.85x(Maximum of burst time amongst all processes)

3.Calculate the threshold value below which if the burst time falls it will be completely executed.
This is given by Threshold=(Max burst time+Min burst time)/2
4.Complete the execution of the first process and push the other processes which have arrived so far into the waiting queue.Then push the first process to the end of waiting queue after decreasing its burst time.
5.For the processes in queue dynamically calculate the time quantum using the formula:
For i in queue
Calculate the sum of remaining burst time of all the i
Find the average
Calculate the time quantum as 0.85Xaverage
6.Now for the processes in queue,begin the execution if the burst time is less than the time quantum.If some process arrives within this period push it into the waiting queue.
7.Repeat the process form 5 to 6 till the waiting queue is not completely empty.

Code:
The code for this algorithm has been attached herewith :

```
#include<iostream>
#include<bits/stdc++.h>
#include<algorithm>


using namespace std;


bool comp(int a, int b)
{
   return (a < b);
}



//comparator function
//compares two pairs according to the arrival time first and then according to their burst times
struct comp1{
   bool operator()(pair<int,int>a,pair<int,int>b)
   {
       if(a.first<b.first)
       return 1;
       else if(a.first==b.first)
       {

if(a.second>b.second)
           return 1;
           return 0;
       }
       else
       return 0;
   }
};



//main
int main()
{
   int n;
   cin>>n;    //number of processes
```

```
   vector<int>a(n);   //arrival
time of processes
   vector<float>b(n);   //burst
time of processes

   //input
   for(int i=1;i<=n;++i)
   {
       cin>>a[i]>>b[i];
   }
```

```
multimap<pair<int,int>,int,comp
1>mp1;//multimap using
comparator function

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

mp1.insert(make_pair(make_pair(
a[i],b[i]),i));
   }
```

```
   vector<int>gnatt;    //gantt
chart
   queue<int>q;
//waiting queue
   set<int>inqueue;
   float rem_time[n];
   float end_time[100];
   float final_time[n];
   int k=0;
   bool finish[n];
   bool visited[n];
   float time_quantum;
   vector<int>temp;

   memset(finish,n,false);
   memset(visited,n,false);
   end_time[0]=0;

   int
max=*max_element(b.begin(),b.en
d());
   int
min=*min_element(b.begin(),b.en
d());
   time_quantum=0.85*max;
   int threshold=(max+min)/2;

   auto u=mp1.begin();

if(b[u->second]>=time_quantum)
   {

gnatt.push_back(u->second);
       k+=1;

end_time[k]=end_time[k-1]+time_
quantum;

rem_time[u->second]=b[u->second
]-time_quantum;

b[u->second]=b[u->second]-time_
quantum;
       visited[u->second]=true;

inqueue.insert(u->second);
   }
   else
   {
```

```
gnatt.push_back(u->second);
        k+=1;

end_time[k]=end_time[k-1]+time_
quantum;
        b[u->second]=0;
        rem_time[u->second]=0;
        finish[u->second]=true;
        visited[u->second]=true;

inqueue.erase(u->second);
    }
    u++;
    while(u!=mp1.end())
    {

if(u->first.first<=time_quantum
)
        {
        q.push(u->second);

inqueue.insert(u->second);
        visited[u->second]=true;
        }
        u++;
    }
    auto it=mp1.begin();
    q.push(it->second);
    visited[it->second]=true;
    inqueue.insert(it->second);



    while(!q.empty())
    {
        int m=q.front();
        cout<<m<<"  ";
        gnatt.push_back(m);
        visited[m]=true;
        inqueue.insert(m);


        int sum=0;
    int n1=q.size();
    int average=0;


    int val;
    while(!q.empty())
    {
        val=q.front();
        temp.push_back(val);
        q.pop();
    }


    for(int
i=0;i<temp.size();++i)
    {
        q.push(temp[i]);
    }


        for(int
v=temp.size()-1;v>=0;--v)
    {
        sum+=b[temp[v]];
    }
    average=sum/n1;
    time_quantum=0.85*average;
```

```
    k+=1;

end_time[k]=end_time[k-1]+time_
quantum;

    float zero=0;
    int temp1;

    if(b[m]>=threshold)
    {

rem_time[m]=b[m]-time_quantum;
        b[m]=rem_time[m];
    }
    else if(b[m]<threshold)
    {
        rem_time[m]=0;
        b[m]=0;
        inqueue.erase(m);
    }

    if(rem_time[m]<=0)
    {
        finish[m]=true;
    }
    else
    {
    finish[m]=false;
    }

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

if((a[i]<=end_time[k])&&(finish
[i]==false)&&(b[i]<=time_quantu
m)&&(inqueue.find(i)==inqueue.e
nd())&&(visited[i]==false))
        {
            q.push(i);
            cout<<i;
            visited[i]=true;
            inqueue.insert(i);
        }
    }

    if(b[m]>0)
    {
    q.push(m);
    }

    temp.clear();
        q.pop();
    }

    for(int
i=0;i<gnatt.size();++i)
    {
        cout<<gnatt[i];
    }
return 0;
}
```

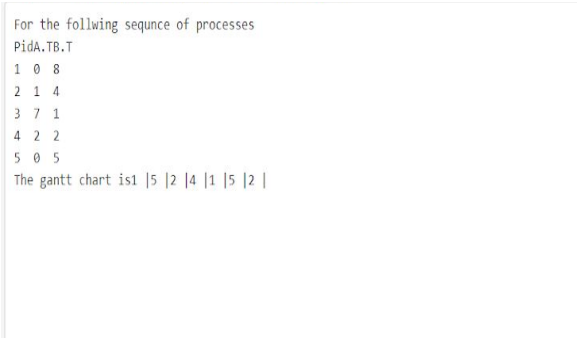
Experimentation:

The experimentation included calculating the number of times the algorithm switches context and comparing it with the number of times the original round robin will change context.Also it has been found that processes are executed for a higher duration than it would have for the original round robin scheduling.
For example:
Consider the following case study.

Process ID	Arrival Time	Burst Time
1	0	8
2	1	4
3	7	1
4	2	2
5	0	5

In an ideal round robin scheduling format the number of context switching would ideally be 10 with a time quantum of 2 units per process(nearly Burst time/Time quantum).

However with our suggested time quantum calculation method the context switching reduced to 8.
This can be reinforced with the following experimental result:



This can be extended to many other different cases and in all those cases this round robin scheduling algorithm either performs the same or better than them.

Results and Discussion:

This algorithm gave improvements over the context switching capabilities which can be improved for Round Robin Scheduling algorithms.Also this algorithm gave preference to the fact that using the fixed time quantum in real world is not a feasible task.Instead using a dynamic time quantum along with threshold values give significantly better results.
For example in the above case this gave an improvement of 20% over the previous round robin scheduling.In real world where the time quantum happens to be much higher this algorithm is bound to perform better simply on the merit of its dynamic nature and its ability to cope with different burst times dynamically.Also the use of threshold makes sure that the algorithms are properly executed within the given time frame and don't go into deadlock situations which would happen during dynamic time calculations.

Conclusion:

This paper suggests a novel dynamic round robin scheduling algorithm which performs better than the previously done works.Future works might include the proper exclusion of outliers which can come during the calculation of threshold and time quantum as it makes use of the maximum and minimum burst time.The threshold values can be better calculated rather than using a average so that the weightage of outliers decreases.

References:

1.“Operating System: A Design-oriented Approach” by Charles Crowley,
2. “Operating Systems: A Modern Perspective” by Gary J Nutt,
3.“Operating System Concepts:” by Silberschatz Galvin Gagne,
4."Operating Systems (2003)" by Deitel, Deitel, and Choffnes.
5.”A new Improved Round Robin-Based Scheduling Algorithm-A comparative Analysis”**DOI:** 10.1109/ICCISci.2019.8716476
6.”Smart Round Robin CPU Scheduling Algorithm For Operating Systems”
DOI: 10.1109/ICEECCOT46775.2019.9114602
7.”ERRA”Efficient Round Robin Algorithm
DOI: 10.1109/CISCT.2019.8777401
8.”Improved Round Robin CPU Scheduling”
DOI: 10.1109/ICGTSPICC.2016.7955294
9.Improved Dynamic Time Slicing Round Robin
DOI: 10.1109/EICT.2017.8275219
10.Smart Round Robin
DOI: 10.1109/ICEECCOT46775.2019.9114602