Southeast University

Lab Final Exam Spring - 22

Title: A comparative study between Round Robin and Priority Scheduling Algorithms

Course: Operating systems Lab

Course Code: CSE3032

Section: 01 (Monday)

Submitted By: Saleh Ibne Omar

ID: 2017000000040

Batch: 45

Semester: Spring 2022

Date of Submission: 21 June, 2022

Title:

A comparative study between Round Robin and Priority Scheduling Algorithms.

Introduction:

Process scheduling is a vital mechanism of Operating Systems, OS schedule process to make the most out of all the running programs in the system. There are several Process Scheduling Algorithms e.g. First Come First Serve, Shortest Job First, Shortest Remaining Job First, Round Robin etc. are the most common of them. For this lab final exam, we will only work with Round Robin (RR) and Priority Scheduling (PS) scheduling algorithms.

Round Robin Scheduling Algorithm:

Round Robin is a Pre-emptive process scheduling algorithm, Round Robin uses regular Queue Data Structure to store processes, same process may occur in ready queue multiple times in Round Robin depending on the size of Time Quantum/Slice.

Time Quantum is a vital variable of Round Robin algorithm because it decides how many time a single process will run after getting CPU until the OS suspends its execution and give other processes opportunity to run for the same amount of time, in this mechanism the running process gets suspended and then gets reinserted in the ready queue until its burst time reaches 0.

Priority Scheduling Algorithm:

Priority scheduling algorithm executes process on the given priority of the process, it uses priority queue to store processes based on highest priority given by the user/system, if a process has more priority over other processes then it will get the CPU first and execute till its burst time reaches 0. Priority scheduling can be two types preemptive and non-preemptive, the priority can be set to ascending or descending order based on the system requirement, for this lab final we used non-preemptive Priority Scheduling and the order or priority is in descending order.

Coding Implementation:

As it is mentioned in the question paper that our computer is Hybrid where it has two CPUs, CPU 1 runs Round Robin scheduling if the priority is equal to 1 and CPU 2 runs Priority scheduling if the priority of the process is greater than 1, so our single code contains both of the algorithms simultaneously based on the conditions.

Code:

```
#include <bits/stdc++.h>
using namespace std;

float TOTAL_TAT_RR = 0.0;
float TOTAL_WT_RR  = 0.0;
float TOTAL_TAT_PS = 0.0;
float TOTAL_WT_PS  = 0.0;


int timeQuantum = 3;


struct Process{
    int ID,AT,BT,WT,CT,TAT,priority,remTime;


    void printDetails(){
        cout<<"PID  = "<<ID
            <<", AT  = "<<AT
            <<", BT  = "<<BT
            <<", WT  = "<<WT
            <<", CT  = "<<CT
            <<", TAT = "<<TAT<<endl;
```

```
    }
};

struct cmp {
    bool operator () (Process const& p1, Process const& p2)
    {
        return p1.priority < p2.priority;
    }
};

Process processArr[1000];
vector<Process> processTime[1000];
queue<Process> readyQRR;
priority_queue<Process, vector<Process>, cmp > readyQPS;
Process currCpuProcessRR;
Process currCpuProcessPS;
bool cpuIsBusyRR = false;
bool cpuIsBusyPS = false;

int main(){

    int numOfProcess;
    int TQ = timeQuantum;
```

```
int countRR = 0, countPS = 0;

freopen("input.txt","r",stdin);
//freopen("output.txt","w",stdout);

cin>>numOfProcess;

    for(int i=1; i<=numOfProcess; i++){
    Process p;
    cin>>p.AT>>p.BT>>p.priority;

    p.ID      = i;
    p.remTime = p.BT;

    processArr[i] = p;
    processTime[p.AT].push_back(processArr[i]);
}

for(int i=0, ct=1; i<1000; i++, ct++){

    if(processTime[i].size()!=0){
        for(int j=0; j<processTime[i].size(); j++){
```

```
        if(processTime[i][j].priority == 1){
            readyQRR.push(processTime[i][j]);
            countRR++;
        }
        else if(processTime[i][j].priority>1){
            readyQPS.push(processTime[i][j]);
            countPS++;
        }
    }
}

if(!readyQRR.empty() && !cpuIsBusyRR){
    currCpuProcessRR = readyQRR.front();
    readyQRR.pop();
    cpuIsBusyRR = true;
    TQ = timeQuantum;
}

if(cpuIsBusyRR){
    currCpuProcessRR.remTime--;
    TQ--;

    if(TQ==0 && currCpuProcessRR.remTime!=0){
```

```
                readyQRR.push(currCpuProcessRR);
                cpuIsBusyRR = false;
            }

            if(currCpuProcessRR.remTime==0){
                currCpuProcessRR.CT  = ct;
                currCpuProcessRR.TAT = currCpuProcessRR.CT  -
currCpuProcessRR.AT;
                currCpuProcessRR.WT  = currCpuProcessRR.TAT -
currCpuProcessRR.BT;
                processArr[currCpuProcessRR.ID] =
currCpuProcessRR;
                TOTAL_TAT_RR+=currCpuProcessRR.TAT;
                TOTAL_WT_RR+=currCpuProcessRR.WT;
                cpuIsBusyRR = false;
            }
        }

        if(!readyQPS.empty() && !cpuIsBusyPS){
            currCpuProcessPS = readyQPS.top();
            readyQPS.pop();
            cpuIsBusyPS = true;
        }
```

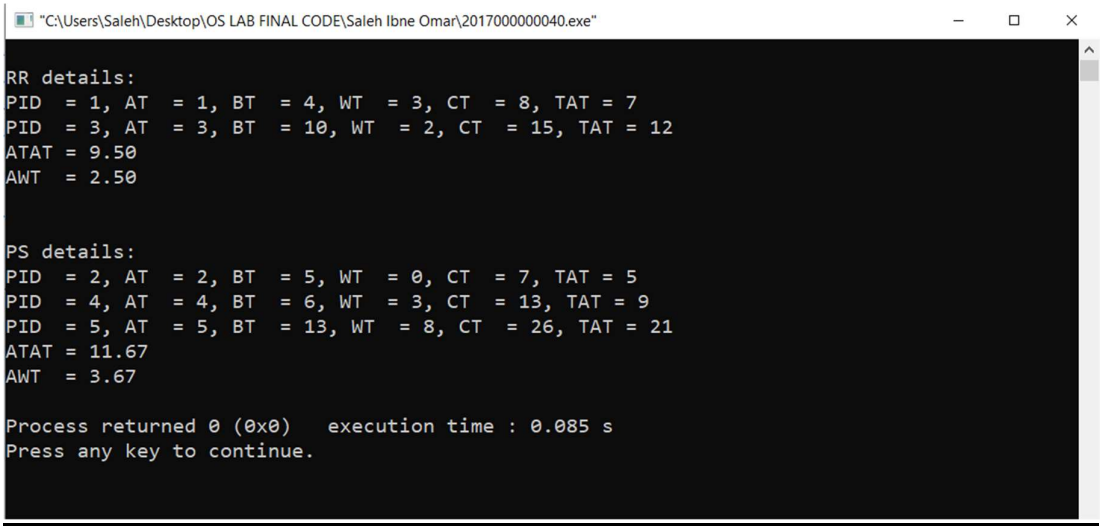
```
        if(cpuIsBusyPS){
            currCpuProcessPS.remTime--;

            if(currCpuProcessPS.remTime==0){
                currCpuProcessPS.CT  = ct;
                currCpuProcessPS.TAT = currCpuProcessPS.CT  -
currCpuProcessPS.AT;
                currCpuProcessPS.WT  = currCpuProcessPS.TAT -
currCpuProcessPS.BT;
                processArr[currCpuProcessPS.ID] =
currCpuProcessPS;
                TOTAL_TAT_PS+=currCpuProcessPS.TAT;
                TOTAL_WT_PS+=currCpuProcessPS.WT;
                cpuIsBusyPS = false;
            }
        }

    }


    cout<<"\nRR details: "<<endl;
    for(int i=1; i<=numOfProcess; i++){
        if(processArr[i].priority == 1){
            processArr[i].printDetails();
```

```
        }
    }

    cout<<"ATAT =
"<<setprecision(2)<<fixed<<(TOTAL_TAT_RR/countRR)<<endl;
    cout<<"AWT  =
"<<setprecision(2)<<fixed<<(TOTAL_WT_RR/countRR)<<endl;

    cout<<"\n\nPS details: "<<endl;
    for(int i=1; i<=numOfProcess; i++){
        if(processArr[i].priority > 1){
            processArr[i].printDetails();
        }
    }

    cout<<"ATAT =
"<<setprecision(2)<<fixed<<(TOTAL_TAT_PS/countPS)<<endl;
    cout<<"AWT  =
"<<setprecision(2)<<fixed<<(TOTAL_WT_PS/countPS)<<endl;

    return 0;
}
```

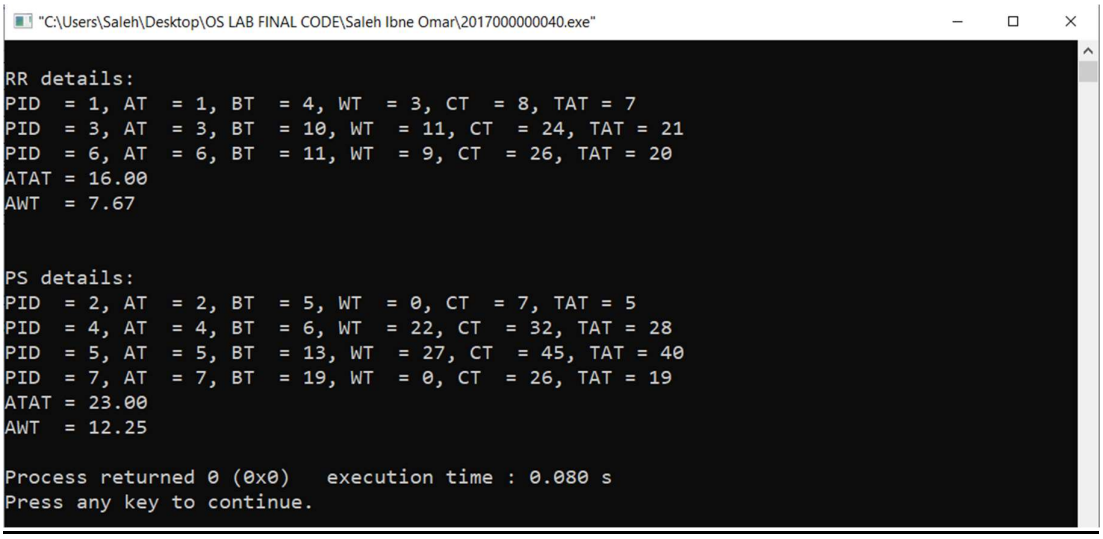
Output Screenshots:

Input taken from text file.

1.

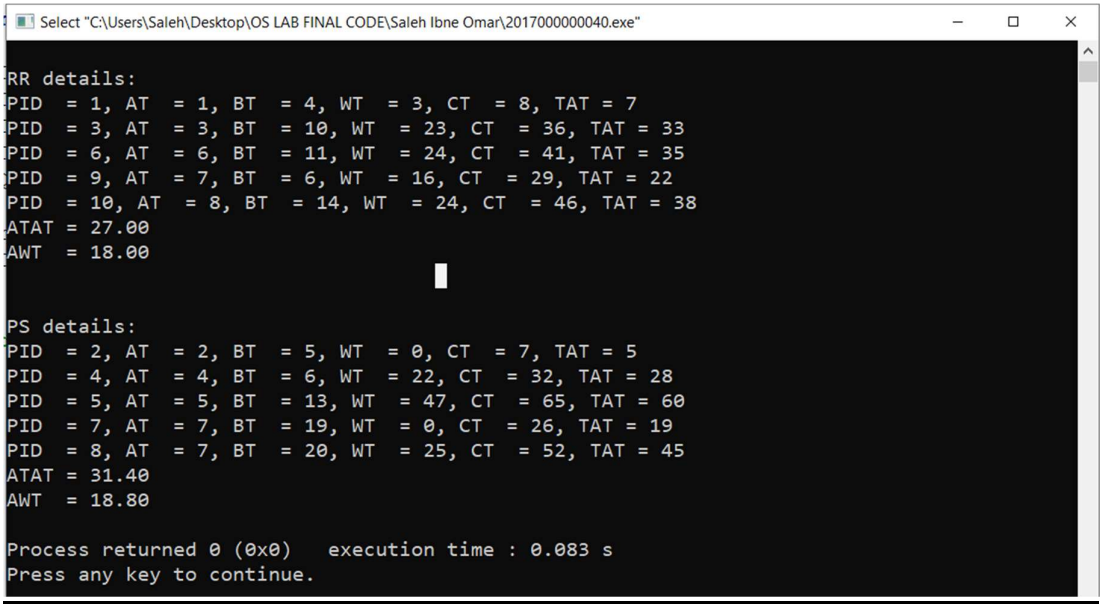


2.



P.T.O

3.



Output observations of RR and PS:

Round Robin		
	ATAT	AWT
TEST CASE 1	9.50	2.50
TEST CASE 2	16.00	7.67
TEST CASE 3	27.00	18.00

Priority Scheduling		
	ATAT	AWT
TEST CASE 1	11.67	3.67
TEST CASE 2	23.00	12.25
TEST CASE 3	31.40	18.80

Comparison of Round Robin and Priority Scheduling Algorithms:

As we can see from the outputs the Round Robin performing slightly better than the Priority scheduling algorithm, this happened because the Round Robin doesn't let any processes to starve, each of the process gets CPU after a particular amount of time which is known as time quantum, in round robin the burst time of a process doesn't matter because each of the process gets CPU in a circular motion.

The Priority scheduling is not performing better than Round robin algorithm because it works on given priority, if a process with highest priority has the most burst time of all it will take long time to execute and other processes will have to wait for that period of time, which causes temporary starvation.

Conclusion:

To compare scheduling algorithms average waiting time and average turnaround is commonly used, in this lab final exam we have also compared ATAT and AWT of the above mentioned algorithms, in comparison with the ATAT and AWT of the RR and PS we can write PS < RR, where RR slightly out performs PS by having the perk of avoiding process starvation.