BCSE – 3rd year – 1st Semester – 2021

Assignment -II

Operating Systems Laboratory

Name: Koustav Dhar Class: BCSE UG-III Roll No.: 001910501022

Problem Statement:

Design a CPU scheduler for jobs whose execution profiles will be in a file that is to be read and an appropriate scheduling algorithm to be chosen by the scheduler.

Format of the profile:

<Job id> <priority> <arrival time> <CPU burst(1) I/O burst(1) CPU burst(2) ..... > -1

(Each information is separated by blank space and each job profile ends with -1. Lesser priority number denotes higher priority process with priority number 1 being the process with the highest priority.)

Example: 2 3 4 100 2 200 3 25 -1 7 1 8 60 10 ..... -1 etc.

Testing:

a. **Create job profiles for 30 jobs and use three different scheduling algorithms (FCFS, Priority, and Round Robin (time slice: 25)).**
b. **Compare the average waiting time, turnaround time of each process for the different scheduling algorithms.**

Screenshot:

For a smaller test case consisting of 3 jobs.

```
kdjonty@KDJonty-Ubuntu-20:~/CSE/Sem5/OS/Lab/Asgn2/1$ ./prob1
Test on randomly generated data? : (y/n)
n
New Job Extracted - Job ID: 1.
New Job Extracted - Job ID: 2.
New Job Extracted - Job ID: 3.
New Job Extracted - Job ID: 1.
New Job Extracted - Job ID: 2.
New Job Extracted - Job ID: 3.
New Job Extracted - Job ID: 1.
New Job Extracted - Job ID: 2.
New Job Extracted - Job ID: 3.
Gantt Chart(FCFS):
{1 @ 0}
{3 @ 30}
{2 @ 32}
{1 @ 35}
{3 @ 39}
{2 @ 43}
Gantt Chart(Round Robin):
{1 @ 0}
{3 @ 25}
{2 @ 27}
{1 @ 30}
{3 @ 35}
{2 @ 39}
{1 @ 42}
Gantt Chart(Priority):
{1 @ 0}
{3 @ 25}
{2 @ 27}
{3 @ 30}
{2 @ 34}
{1 @ 37}
{1 @ 47}
Analysis for FCFS Scheduling Algorithm:
Average Turnaround Time = 42.3333
Average Waiting Time = 23
Analysis for Round Robin Scheduling Algorithm:
Average Turnaround Time = 42
Average Waiting Time = 22.6667
Analysis for Priority Based Scheduling Algorithm:
Average Turnaround Time = 40.3333
Average Waiting Time = 21
kdjonty@KDJonty-Ubuntu-20:~/CSE/Sem5/OS/Lab/Asgn2/1$
```

For a test case generated consisting of 30 jobs.

```
{1 @ 3463}
{14 @ 3475}
{14 @ 3479}
{14 @ 3504}
{25 @ 3511}
{17 @ 3519}
{25 @ 3544}
{17 @ 3546}
{14 @ 3575}
{17 @ 3585}
{14 @ 3593}
{14 @ 3596}
{25 @ 3610}
{25 @ 3646}
{25 @ 3671}
{25 @ 3696}
{17 @ 3710}
{25 @ 3714}
{25 @ 3739}
{17 @ 3746}
{25 @ 3771}
{17 @ 3772}
{25 @ 3779}
{25 @ 3804}
{25 @ 3829}
{17 @ 3843}
{17 @ 3867}
{25 @ 3875}
{25 @ 3900}
Analysis for FCFS Scheduling Algorithm:
Average Turnaround Time = 2506.53
Average Waiting Time = 2264.6
Analysis for Round Robin Scheduling Algorithm:
Average Turnaround Time = 2487.9
Average Waiting Time = 2245.97
Analysis for Priority Based Scheduling Algorithm:
Average Turnaround Time = 2171.23
Average Waiting Time = 1929.3
kdjonty@KDJonty-Ubuntu-20:~/CSE/Sem5/OS/Lab/Asgn2/1$
```

Commented Code:

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

// class to store details of a job, including their execution profiles
class Job {
private:
   int jobId;                  // job id
   int priority;               // priority (lesser value, higher priority)
   int arrivalTime;            // arrival time
```

```
   vector<int> cpuBursts;      // patches of cpu burst times, i.e. cpu burst(1), cpu
burst(2), ...
   vector<int> ioBursts;       // patches of i/o burst times, i.e. i/o burst(1), i/o
burst(2), ...
   int cntCPU;                 // total count of cpu bursts to occur
   int cntIO;                  // total count of i/o bursts to occur
   int nextCPU;                // index of next cpu burst to be performed
   int nextIO;                 // index of next i/o burst to be performed
   int nextArrivalTime;        // next arrival time for ready queue
   int totalTimeReqd;          // sum of all bursts
   bool preempt;               // to track flag for preemptive algorithms
   // the execution profile of the job is like: cpu burst(1), i/o burst(1), cpu
burst(2), i/o burst(2), ...
public:
   Job() {                     // default constructor
       jobId = priority = arrivalTime = cntCPU = cntIO = nextCPU = nextIO =
nextArrivalTime -1;
   }
   Job(vector<int> execProfile) {  // paramaterized constructor with execution profile
as the argument
       int sz = execProfile.size();
       // for (int i = 0; i < sz; ++i)
       //     cout << execProfile[i] << " ";
       // cout << "\n";
       jobId = execProfile[0];
       priority = execProfile[1];
       nextArrivalTime = arrivalTime = execProfile[2];
       cntCPU = cntIO = 0;
       totalTimeReqd = 0;
       preempt = false;
       for (int i = 3; i < sz; ++i) {
           if (i & 1) {        // for cpu bursts
               cpuBursts.push_back(execProfile[i]);
               ++cntCPU;
           } else {            // for i/o bursts
               ioBursts.push_back(execProfile[i]);
               ++cntIO;
           }
           totalTimeReqd += execProfile[i];
       }
       nextCPU = nextIO = 0;
       cout << "New Job Extracted - Job ID: " << jobId << ".\n"; // debug statement
   }
   // getter functions
   int getJobId()          {   return jobId;           }
   int getPriority()       {   return priority;        }
   int getArrivalTime()    {   return arrivalTime;     }
```

```
    int getCntCPU()          {   return cntCPU;          }
    int getCntIO()           {   return cntIO;           }
    int getNextCPU()         {   return nextCPU;         }
    int getNextIO()          {   return nextIO;          }
    int getCurrCPUTime()     {   return cpuBursts[nextCPU];}
    int getCurrIOTime()      {   return ioBursts[nextIO];}
    int getNextArrivalTime(){   return nextArrivalTime; }
    int getTotalTime()       {   return totalTimeReqd;   }
    int getPreempt()         {   return preempt;         }
    // setter functions
    void incNextCPU()        {   if (nextCPU < cntCPU)   ++nextCPU;  }
    void incNextIO()         {   if (nextIO < cntIO)     ++nextIO;   }
    void setPreempt()        {   preempt = true;                     }
    void unsetPreempt()      {   preempt = false;                    }
    void updateCPUTime(int dur) {
        cpuBursts[nextCPU] -= dur;
    }
    void updateArrival(int dur) {
        if (cpuBursts[nextCPU] == dur)
            nextArrivalTime = (cpuBursts[nextIO] + ioBursts[nextIO]);
        cpuBursts[nextCPU] -= dur;
    }
    // checker functions
    bool cpuLeft()           {   return nextCPU < cntCPU;    }
    bool ioLeft()            {   return nextIO < cntIO;      }

};

// comparator class for ordering jobs on the basis of arrival time
class JobComparatorFCFS {
public:
    bool operator()(Job& a, Job& b) {
        return a.getArrivalTime() < b.getArrivalTime();
    }
};

// comparator class for ordering jobs on the basis of priority
class JobComparatorPriority {
public:
    bool operator()(Job& a, Job& b) {
        return a.getPriority() > b.getPriority();
    }
};

// abstract class for Job Scheduling algorithms
class JobScheduler {
protected:
```

```
    vector<Job> jobs;
    int jobsLeft;
    int totalWaitingTime;
    float avgWaitingTime;
    int totalTurnaroundTime;
    float avgTurnaroundTime;
public:
    // to parse the file and create vector of job profiles
    JobScheduler(string filename) {
        ifstream fin;
        fin.open(filename, ios::in);
        int num;
        totalTurnaroundTime = totalWaitingTime = jobsLeft = 0;
        while (fin >> num) {
            if (num == -1) {
                break;
            } else {    // new job starting
                vector<int> v = {num};
                fin >> num;
                if (num != -1)
                    v.push_back(num);
                while (fin >> num) {
                    if (num == -1) {
                        break;
                    } else {
                        v.push_back(num);
                    }
                }
                Job J(v);
                jobs.push_back(J);
                ++jobsLeft;
            }
        }
    }
    // pure virtual function to schedule processes following the scheduling algorithms
    virtual void schedule() = 0;
    // to show results like average waiting and turnaround time
    void showAnalysis() {
        avgWaitingTime = totalWaitingTime * 1.0 / jobs.size();
        avgTurnaroundTime = totalTurnaroundTime * 1.0 / jobs.size();
        cout << "Average Turnaround Time = " << avgTurnaroundTime << "\n";
        cout << "Average Waiting Time = " << avgWaitingTime << "\n";
    }
};

// class for fcfs scheduling, inherited from JobScheduler class
class FCFS_Scheduler: public JobScheduler {
```

```
    queue<Job> ready_queue; // ready queue for CPU
    unordered_map<int, vector<Job>> block_queue; // block queue for i/o operations
    vector<pair<int, int>> ganttChart;  // to store the schedule
public:
    // sort the jobs based on arrival time
    FCFS_Scheduler(string filename): JobScheduler(filename) {
        sort(jobs.begin(), jobs.end(), JobComparatorFCFS());
    }

    virtual void schedule() {
        int sz = jobs.size(), index = 0;
        for (int i = 0; i < sz; ++i) {  // push all the processes initially into the
block queue
            block_queue[jobs[i].getArrivalTime()].push_back(jobs[i]);
            totalTurnaroundTime += jobs[i].getTotalTime();
            // cout << jobs[i].getJobId() << " -> " << jobs[i].getArrivalTime() <<
"\n";
        }
        int timeline = 0;
        bool cpuEmpty = true;
        Job currentJob; int nextTerminate = INT_MAX;
        while (jobsLeft > 0) {
            // process coming from block queue
            for (Job j : block_queue[timeline]) {
                ready_queue.push(j);
            }
            if (timeline == nextTerminate) {
                cpuEmpty = true;
                if (currentJob.cpuLeft() == false)
                    --jobsLeft;
            }
            if (cpuEmpty) {
                if (!ready_queue.empty()) {
                    currentJob = ready_queue.front();
                    ready_queue.pop();
                    nextTerminate = timeline + currentJob.getCurrCPUTime();
                    ganttChart.push_back({currentJob.getJobId(), timeline});
                    cpuEmpty = false;
                    currentJob.updateArrival(currentJob.getCurrCPUTime());
                    currentJob.incNextCPU();
                    if (currentJob.ioLeft()) {
                        currentJob.incNextIO();
                        if (currentJob.cpuLeft()) {

block_queue[timeline+currentJob.getNextArrivalTime()].push_back(currentJob);
                        }
                    }
```

```
            }
        }
        // cout << timeline << " " << ready_queue.size() << "\n";
        totalTurnaroundTime += ready_queue.size();
        totalWaitingTime += ready_queue.size();
        ++timeline;
    }
    cout << "Gantt Chart(FCFS): \n";
    for (pair<int, int> t : ganttChart) {
        cout << "{" << t.first << " @ " << t.second << "}\n";
    }
  }
  void showAnalysis() {
    cout << "Analysis for FCFS Scheduling Algorithm: \n";
    JobScheduler::showAnalysis();
  }
};

// class for round robin scheduling, inherited from JobScheduler class
class RoundRobin_Scheduler: public JobScheduler {
  queue<Job> ready_queue; // ready queue for CPU
  unordered_map<int, vector<Job>> block_queue; // block queue for i/o operations
  vector<pair<int, int>> ganttChart;  // to store the schedule
  int timeSlice;
public:
  // sort the jobs based on arrival time
  RoundRobin_Scheduler(string filename): JobScheduler(filename) {
    sort(jobs.begin(), jobs.end(), JobComparatorFCFS());
    timeSlice = 25;
  }
  virtual void schedule() {
    int sz = jobs.size(), index = 0;
    for (int i = 0; i < sz; ++i) {  // push all the processes initially into the
block queue
        block_queue[jobs[i].getArrivalTime()].push_back(jobs[i]);
        totalTurnaroundTime += jobs[i].getTotalTime();
        // cout << jobs[i].getJobId() << " -> " << jobs[i].getArrivalTime() <<
"\n";
    }
    int timeline = 0;
    bool cpuEmpty = true;
    Job currentJob; int nextTerminate = INT_MAX;
    while (jobsLeft > 0) {
        // process coming from block queue
        for (Job j : block_queue[timeline]) {
            ready_queue.push(j);
        }
```

```
            if (timeline == nextTerminate) {
                cpuEmpty = true;
                if (currentJob.cpuLeft() == false)
                    --jobsLeft;
                else if (currentJob.getPreempt() == true && currentJob.getCurrCPUTime()
> 0) {
                    currentJob.unsetPreempt();
                    ready_queue.push(currentJob);
                }
            }
            if (cpuEmpty) {
                if (!ready_queue.empty()) {
                    currentJob = ready_queue.front();
                    ready_queue.pop();
                    ganttChart.push_back({currentJob.getJobId(), timeline});
                    cpuEmpty = false;
                    int val = min(currentJob.getCurrCPUTime(), timeSlice);
                    nextTerminate = timeline + val;
                    currentJob.updateArrival(val);
                    if (currentJob.getCurrCPUTime() == 0) {
                        currentJob.incNextCPU();
                        if (currentJob.ioLeft()) {
                            currentJob.incNextIO();
                            if (currentJob.cpuLeft()) {

block_queue[timeline+currentJob.getNextArrivalTime()].push_back(currentJob);
                            }
                        }
                    } else {
                        currentJob.setPreempt();
                    }
                }
            }
            // cout << timeline << " " << ready_queue.size() << "\n";
            totalTurnaroundTime += ready_queue.size();
            totalWaitingTime += ready_queue.size();
            ++timeline;
        }
        cout << "Gantt Chart(Round Robin): \n";
        for (pair<int, int> t : ganttChart) {
            cout << "{" << t.first << " @ " << t.second << "}\n";
        }

    }
    void showAnalysis() {
        cout << "Analysis for Round Robin Scheduling Algorithm: \n";
        JobScheduler::showAnalysis();
```

```
    }
};

// class for priority based scheduling, inherited from JobScheduler class
class Priority_Scheduler: public JobScheduler {
    priority_queue<Job, vector<Job>, JobComparatorPriority> ready_queue; // ready queue
for CPU
    unordered_map<int, vector<Job>> block_queue; // block queue for i/o operations
    vector<pair<int, int>> ganttChart;  // to store the schedule
    int timeSlice;
public:
    // sort the jobs based on arrival time
    Priority_Scheduler(string filename): JobScheduler(filename) {
        sort(jobs.begin(), jobs.end(), JobComparatorFCFS());
        timeSlice = 25;
    }
    virtual void schedule() {
        int sz = jobs.size(), index = 0;
        for (int i = 0; i < sz; ++i) {  // push all the processes initially into the
block queue
            block_queue[jobs[i].getArrivalTime()].push_back(jobs[i]);
            totalTurnaroundTime += jobs[i].getTotalTime();
            // cout << jobs[i].getJobId() << " -> " << jobs[i].getArrivalTime() <<
"\n";
        }
        int timeline = 0;
        bool cpuEmpty = true;
        Job currentJob; int nextTerminate = INT_MAX;
        while (jobsLeft > 0) {
            // process coming from block queue
            for (Job j : block_queue[timeline]) {
                ready_queue.push(j);
            }
            if (timeline == nextTerminate) {
                cpuEmpty = true;
                if (currentJob.cpuLeft() == false)
                    --jobsLeft;
                else if (currentJob.getPreempt() == true && currentJob.getCurrCPUTime()
> 0) {
                    currentJob.unsetPreempt();
                    ready_queue.push(currentJob);
                }
            }
            if (cpuEmpty) {
                if (!ready_queue.empty()) {
                    currentJob = ready_queue.top();
                    ready_queue.pop();
```

```
                    ganttChart.push_back({currentJob.getJobId(), timeline});
                    cpuEmpty = false;
                    int val = min(currentJob.getCurrCPUTime(), timeSlice);
                    nextTerminate = timeline + val;
                    currentJob.updateArrival(val);
                    if (currentJob.getCurrCPUTime() == 0) {
                        currentJob.incNextCPU();
                        if (currentJob.ioLeft()) {
                            currentJob.incNextIO();
                            if (currentJob.cpuLeft()) {

block_queue[timeline+currentJob.getNextArrivalTime()].push_back(currentJob);
                            }
                        }
                    } else {
                        currentJob.setPreempt();
                    }
                }
            }
            // cout << timeline << " " << ready_queue.size() << "\n";
            totalTurnaroundTime += ready_queue.size();
            totalWaitingTime += ready_queue.size();
            ++timeline;
        }
        cout << "Gantt Chart(Priority): \n";
        for (pair<int, int> t : ganttChart) {
            cout << "{" << t.first << " @ " << t.second << "}\n";
        }

    }
    void showAnalysis() {
        cout << "Analysis for Priority Based Scheduling Algorithm: \n";
        JobScheduler::showAnalysis();
    }
};

// class to run the simulation
class Runner {
    string filename;
public:
    Runner(string f) {
        filename = f;
    }
    void filegenerator() {
        int sz = 30;
        string filename = "jobprofiles_random.txt";
        ofstream fout;
```

```
        fout.open(filename, ios::out);
        for (int i = 1; i <= sz; ++i) {
            fout << i << " ";
            int arrival = rand() % 100;
            fout << arrival << " ";
            int priority = rand() % 17;
            fout << priority << " ";
            int burstsz = 1 + rand() % 13;
            for (int j = 0; j < burstsz; ++j) {
                // cpu
                int exp = rand() % 7;
                fout << (1 << exp) << " ";
                // i/o
                exp = rand() % 7;
                fout << (1 << exp) << " ";
            }
            int last = rand() % 2;
            if (last) {
                int exp = rand() % 7;
                fout << (1 << exp) << " ";
            }
            fout << "-1 ";
        }
    }
    void run() {
        cout << "Test on randomly generated data? : (y/n)\n";
        char choice;
        cin >> choice;
        if (choice == 'y' || choice == 'Y') {
            filename = "jobprofiles_random.txt";
            filegenerator();
        }
        FCFS_Scheduler F(filename);
        RoundRobin_Scheduler R(filename);
        Priority_Scheduler P(filename);
        F.schedule();
        R.schedule();
        P.schedule();
        F.showAnalysis();
        R.showAnalysis();
        P.showAnalysis();
    }
};


signed main() {
```

```
    Runner R("jobprofiles_random.txt");
    R.run();

    return 0;
}
```