Scheduling Algorithms Simulations

Kathmandu University
School of Science

Author	Manish Pokharel
Professor	Dr. Sudhan Jha
Batch	CS-21
Roll No.	42
Submitted on	April 9, 2024

1 Introduction

In this report, a set of 5 processes were put through 3 different Scheduling Algorithms, namely:

- First-Come, First-Served
- Shortest Job First - Non Preemptive
- Shortest Remaining Time First - Preemptive

The average waiting time and the average turn around time were then calculated as the com- parison matrices. Similarly, Gantt Charts depicting the execution timeline of each algorithm were also constructed. The advantages and disadvantages of each algorithms were also discussed along with which may be suitable for each scenario.

The set of processes put through the algorithms is:

Process:	Arrival Time:	Burst Time:
P1	0	10
P2	2	15
P3	02	5
P4	05	20
P5	055	10

2 GANTT Charts

Below are the Gantt Charts of the 3 Scheduling Algorithms:

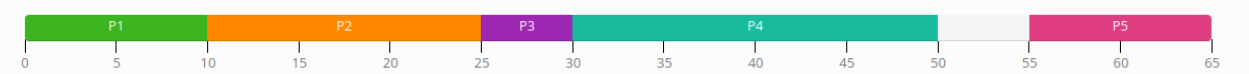

Figure 1: Gantt Chart of FCFS

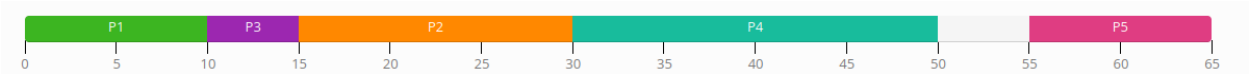

Figure 2: Gantt Chart of SJF

Figure 3: Gantt Chart of SJRF

3 Performance

Algorithm	Avg. WT	Avg. TAT
FCFS	11.2	23.2
SJF	9.2	21.2
SRJF	8.6	20.6

Figure 4: Performance of Algorithms

From the above table, it is apparent that the most efficient algorithm for this particu- lar set of processes was the SJRF algorithm, followed closely by the SJF algorithm, and then the FCFS algorithm.

4 Advantages and Disadvantages

4.1 First-Come, First-Served

4.1.1 Advantages

- **Ease of implementation**: FCFS is simple to understand and put into practice, making it accessible for programmers and system designers.
- **Starvation avoidance**: Since processes are executed based on their arrival order, there is no risk of starvation. Each process eventually receives CPU time, albeit possibly after a prolonged wait.
- **Low overhead**: FCFS scheduling entails minimal overhead as it does not require intricate calculations or decision-making processes.

4.1.2 Disadvantages

- **Convoy effect**: FCFS can result in the convoy effect, wherein short processes are delayed behind lengthy ones, leading to inefficient CPU utilization.
- **Subpar average response time**: FCFS may yield suboptimal average response time, particularly if lengthy processes are prioritized first. This can create a perception of inadequate system performance.
- **Lack of process priority consideration**: FCFS overlooks process priority, potentially causing critical or high-priority processes to be deferred in favor of less significant ones.

4.2 Shortest Job First - Non Preemptive

4.2.1 Advantages

- **Optimization for average waiting time**: SJF non-preemptive scheduling minimizes average waiting time by prioritizing shorter jobs, resulting in quicker process turnaround.
- **Absence of convoy effect**: Unlike FCFS, SJF non-preemptive scheduling avoids the convoy effect, ensuring efficient CPU utilization without short processes being hindered by long ones.
- **Predictability**: With advance knowledge of burst times for all processes, the scheduler can make informed decisions, leading to predictable scheduling behavior.

4.2.2 Disadvantages

- **Potential for starvation**: Longer jobs may face starvation if a continuous stream of short jobs arrives, as shorter jobs always take precedence, potentially indefinitely delaying execution of longer processes.
- **Requirement for burst time prediction**: SJF non-preemptive scheduling relies on accurate prediction of burst times for each process, which may be impractical in systems with varying workload characteristics.
- **Risk of indefinite postponement**: Continuous arrival of shorter jobs may indefinitely postpone longer jobs, resulting in poor responsiveness for those processes.

4.3 Shortest Remaining Time First - Preemptive

4.3.1 Advantages

- **Optimization for average waiting time**: SRTF preemptive scheduling minimizes average waiting time by prioritizing the shortest remaining job, resulting in faster process turnaround compared to other scheduling algorithms.
- **Efficient resource utilization**: SRTF maximizes CPU utilization and minimizes idle time by continually selecting the shortest remaining job for execution, leading to efficient resource utilization.
- **Fairness**: SRTF preemptive scheduling ensures fairness by granting higher priority to shorter jobs, preventing longer processes from monopolizing CPU time and potentially causing shorter processes to experience extended waits.

4.3.2 Disadvantages

- **High context switching overhead**: SRTF preemptive scheduling involves frequent context switches as the currently executing process may be preempted by a shorter job. This can result in high overhead and may reduce overall system performance.
- **Possibility of starvation for longer jobs**: Continuous arrival of shorter jobs may lead to longer jobs being continuously preempted and potentially starved of CPU time, resulting in poor responsiveness for those processes.
- **Unpredictability**: The constant preemption of processes in SRTF preemptive scheduling can make system behavior less predictable as process execution may be interrupted at any time, leading to non-deterministic performance.

5 Possible Scenarios

5.1 Possible Scenarios

Here are some potential scenarios where each of these scheduling algorithms may be optimal:

5.1.1 First-Come, First-Served

Batch Processing: In scenarios prioritizing simplicity and fairness over efficiency, FCFS can be optimal. For example, in batch processing systems aimed at executing numerous similar tasks with minimal variation in execution time, FCFS guarantees tasks are processed in the order of submission, without the need for complex scheduling algorithms.

5.1.2 Shortest Job First

Scientific Computing: In scientific computing environments with significantly varied job execution times, SJF can be optimal. For instance, in computational research labs conducting various simulations, SJF ensures quicker completion of shorter simulations, leading to faster overall turnaround times and improved resource utilization.

5.1.3 Shortest Remaining Time First

Real-Time Systems: In real-time systems where meeting deadlines is paramount, SRTF can be optimal. For example, in systems controlling robotic arms where tasks arrive dynamically and must be processed within specific time constraints, SRTF ensures the shortest remaining task is always executed next, minimizing response time and maximizing system reliability.

6 Conclusion

In this report, we discussed three scheduling algorithms: FCFS, SJF, and SRTF. We compared their performance based on average waiting time and average turnaround time for a set of five processes. The results revealed that, in this particular case, SJF proved to be the most efficient algorithm. Additionally, we examined the advantages and disadvantages of each algorithm, along with scenarios in which they may be optimally applicable.

7 Code Snippets

lab.cpp

```cpp
#include <iostream>
#include <queue>
#include <algorithm>
#include <vector>
#include <climits>

using namespace std;

class process {
    public:
        int ID;
        int arrivaltime;
        int bursttime;
        int remainingtime;
        int starttime;
        int endtime;

        process(int i, int a, int b) {
            ID = i;
            arrivaltime = a;
            bursttime = b;
            remainingtime = b;
            starttime = 0;
            endtime = 0;
        }

          // Function to update remaining time
        void updateRemainingtime(int time) {
            remainingtime = max(0, remainingtime - time);
        }
};

// sorts the processes according to arrival time
void processSorter(vector<process>& processes) {
    for (int i = 0; i<processes.size(); i++) {
        for(int j = i+1; j<processes.size(); j++) {
            if(processes[i].arrivaltime > processes[j].arrivaltime) {
                process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}
```

lab.cpp

```cpp

// sorts the processes according to burst time
void processSorterbyBurst(vector<process>& processes) {
    for (int i = 0; i<processes.size(); i++) {
        for(int j = i+1; j<processes.size(); j++) {
            if(processes[i].bursttime > processes[j].bursttime) {
                process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}


void printprocess(vector<process> processes) {
    cout << " ID \t Arrival time  \t Burst time \t Start time \t End time" << endl;
    for(auto p: processes) {
        cout << p.ID << "\t\t" << p.arrivaltime << "\t\t" << p.bursttime << "\t\t" << p.starttime << "\t\t" << p.endtime << endl;
    }
}


// FCFS scheduler algorithm

void FCFS(vector<process>& processes) {
    processSorter(processes);
```

```cpp
bool compareAT(const process& p1, const process& p2) {
    return p1.arrivaltime < p2.arrivaltime;
}

bool compareRT(const process& p1, const process& p2) {
    return p1.remainingtime < p2.remainingtime;
}

// SJF (Non-preemptive scheduler)
void npSJF(vector<process>& processes) {
    processSorterbyBurst(processes);

    FCFS(processes);
}
```

```cpp
// SJF (pre-emptive scheduler)

vector<process> pSJF(vector<process>& processes) {
    int n = processes.size();
    int currenttime = 0;
    int completed = 0;
    vector<process> readyQueue;
    vector<process> finalProduct;

    processSorter(processes);

    while(completed != n) {
        //iterate through the processes to find out what processes have arrived.
        // if a process whose arrival time = currenttime, add to the ready Queue.

        for(auto p: processes) {
            if (p.arrivaltime = currenttime) readyQueue.push_back(p);
        }

        if(!readyQueue.empty()) {
            processSorterbyBurst(readyQueue);

            // get the first vector
            process temp = readyQueue.front();

            if(temp.arrivaltime > currenttime) {
                temp.starttime = temp.arrivaltime;
                currenttime = temp.arrivaltime;
            } else temp.starttime = currenttime;

            currenttime += temp.bursttime;
            temp.endtime = currenttime;

            finalProduct.push_back(temp);
            readyQueue.erase(readyQueue.begin());
            completed++;
        }

        currenttime++;
    }
    return finalProduct;
}
```

```cpp
float avgWT(vector<process> processes) {
    float totalWT = 0;
    for(auto p: processes) {
        totalWT += (p.starttime - p.arrivaltime);
    }
    return totalWT/processes.size();
}

float avgTAT(vector<process> processes) {
    float totalTAT = 0;
    for(auto p: processes) {
        totalTAT += (p.endtime - p.arrivaltime);
    }
    return totalTAT/processes.size();
}
```

```cpp
int main() {

    vector<process> processes = {
        {1, 0, 10},
        {2, 2, 15},
        {3, 2, 5},
        {4, 5, 20},
        {5, 55, 10}
    };

    vector<process> P1 = processes;
    vector<process> P2 = processes;
    vector<process> P3 = processes;



    FCFS(P1);
    npSJF(P2);
    P3 = pSJF(P3);

    cout << "Algorithm \t Avg. WT \t Avg. TT" << endl;
    cout << "FCFS \t\t" << avgWT(P1) << "\t\t" << avgTAT(P1) << endl;
    cout << "npSJF \t\t" << avgWT(P2) << "\t\t" << avgTAT(P2) << endl;
    cout << "pSJF \t\t" << avgWT(P3) << "\t\t" << avgTAT(P3) << endl;


    return 0;
}
```