Daniel Morgan

Jeff Foster

# Project: Scheduling Simulation

## Intro

The purpose of this project is to demonstrate knowledge of scheduling algorithms learned in the course *Introduction to Operating Systems*. This is accomplished through developing a simulation of three different scheduling algorithms: Round Robin, First In-First Out (FIFO), and Shortest Job First (SJF). After completion of simulator development, tests are then run to verify the accuracy of the project, as well as analyze the performance of said algorithms. These tests are run for both a single processor simulation and a multi-processor system with four available processors. We then use these results to postulate as to the benefits and drawbacks of each.

## Methodology and limitations

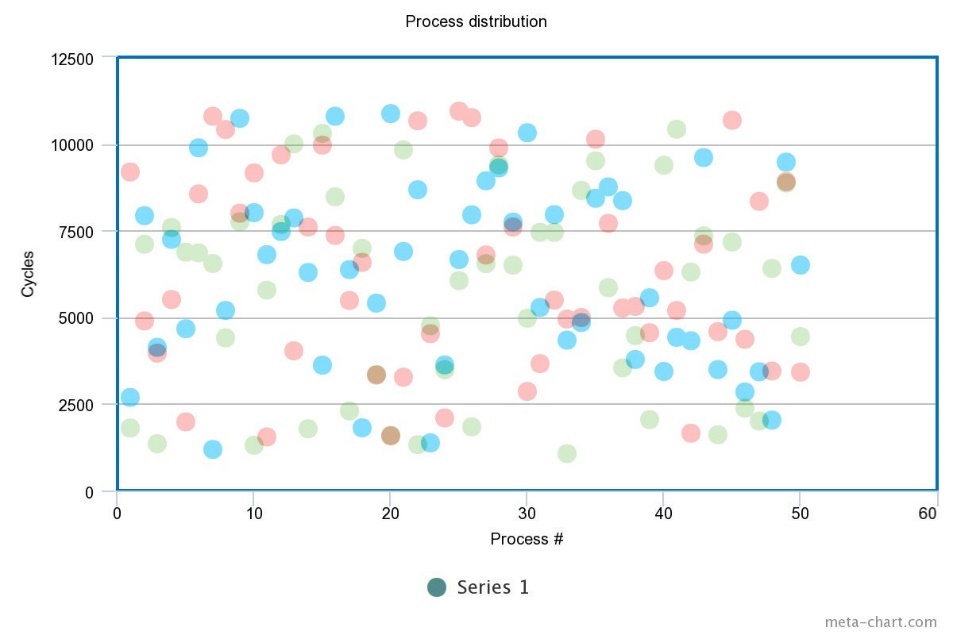
The primary building block for the simulator is a process generator, developed in an earlier homework assignment, which generates a set of fifty unique randomly-generated processes for these simulations. These processes are then utilized by an additional scheduling component developed for this project. Since our team is comprised of two members, we decided to split the tasks amongst each other. Three algorithms don’t divide evenly, so one member developed the Round Robin and FIFO simulation, while the other developed the SJF simulation.

Each member originally developed a unique solution for the basis process generator, one of which was developed in the C language and the other in the C++ language. Each member used his own unique solution to build upon. As such, the solutions for the Round Robin & FIFO simulator, developed in C, and the SJF simulator, developed in C++, have differing compile requirements but accomplish similar goals according to their respective algorithms. Please refer to the project README docs for compile and run requirements.

For the multi-processor portion of the assignment, we broke the processes down into four groups, two sets of thirteen and two sets of twelve, and assigned each set to a single processor. The scheduling algorithm was then executed on each processor set independently, effectively mimicking a four-processor system.

### Round Robin

The Round Robin implementation utilizes a linked list structure to store information about each process, such as process ID, cycle time and arrival time, amongst others. An Integer value was used to represent time, and a for loop was used to perform the Round Robin routine. Three sample cycle time distributions can be seen below, with each color representing a different test:



Each test utilizes a non-uniform distribution of 50 processes in the range of 1000 to 11000 cycles, with a mean cycle time tending towards 6000. We chose to go with a non-uniform distribution of processes, for all programs, to best model a real-world situation, as the required runtime of each process arriving into the ready queue cannot always be known in advance by the schedule or many of the other running processes. Also, because of the dynamic nature of a scheduler, it is highly unlikely that a real world set of processes would be uniform.

The program first checks to see if the process has arrived, and then performs the necessary operations so long as there is still some remaining work that needs to be done in the process queue. It also assumes that a process entering the ready queue the same time as a previous Round Robin iteration has completed will be immediately scheduled and appended. For example, if p1 arrives at cycle 0 and runs for 50 cycles, p2 will run next so long as it arrives at or before the 50-cycle mark.

For testing purposes, the program results were checked by hand to ensure accuracy of results. Because of the considerable number of context switches and processes, ten unique smaller sets were individually tested. These tests were found to always produces the correct results.

### FIFO

The FIFO implementation uses the same linked list structure as the Round Robin implementation. However, it simply appends incoming processes to the end of the queue and runs each process until completion, so long as the process has arrived within the current processor cycle. Like the Round Robin implementation, the FIFO implementation was checked by hand for accuracy with ten unique sets of fifty randomly generated processes.

### SJF

The SJF program is developed in C++ and follows the classical OOP approach allowed by the language. Its method of generating processes uses a non-uniform gaussian distribution of cycle lengths in the 1000 to 11000 range like the RR/FIFO program, and adheres to a mean cycle time tending towards 6000.

The program is comprised of four C++ class components:

* Process

The most basic component - describes and manages a single generated process.

* ProcessGenerator

Contains the logic/methods to randomly generate unique Process objects and stores these processes in a C++ standard library vector, *processes*, to be later distributed amongst the available processors.

* Processor

Represents a single processor called upon by the scheduler with its own unique *cpuID* and properties related to its running cycles and jobs, as well as a *processes* vector containing its available jobs to schedule and run.

The class contains two methods that are crucial to the scheduling algorithm: *findShortestJob* that returns the next job the processor will switch to, and *simulateCycle* used to simulate processing cycles and job executions. Each execution of *simulateCycle* begins with a call to *findShortestJob* to grab the next appropriate job from the ready queue (*processes* vector).

* Scheduler*\_SJF*

Represents the scheduler simulator that ties everything together. Central to the *Scheduler\_SJF* object is the *run* method. To begin, it creates a new *ProcessGenerator* that is pointed to by the member *pGen* and calls the *generate\_batch* method to generate a set of processes. Then a new *Processor is* created for every processor that it will simulate, either single or four-processor, and added to the *processors* vector. The generated processes are then distributed amongst these processors.

Next, it executes a while loop, the core of the scheduling algorithm, that runs until the *validJobsExist* method returns false - meaning all jobs have been completed. At each step of the loop, each available processor calls its *simulateCycle* method to simulate a full run of its current job’s cycles or simulate one processor cycle if no valid jobs have arrived yet within the current cycle.

The SJF simulator’s accuracy was tested in development by using a handful of smaller, known sets of processes instead of a set of fifty random processes. In this way, the outcome was pre-determined and the simulation’s results were easily verified against the known results. This was done initially against a single processor simulation, and once results were verified this set was then tested against the multi-processor system and verified.

Knowing that each simulator produced dependable and easily verifiable results allowed the team to move forward with the experiment using several randomly generated sets of fifty unique processes, of which the experiment’s corresponding data is provided in the next section.

## Experiment Results

Below is the data gathered from experiments run using both single processor and multi-processor system simulations. Our goal was to collect and examine data related to average wait times and total context-switch penalties. The Context switch penalty charts only display Round Robin results because both FIFO and SJF always give a penalty of 500 cycles for any single processor trial, and 470 for any four-processor trial.

|  |  |
| --- | --- |
| Figure 1: Avg. Wait Time - single processor (10 trials) | Figure 2: RR Context-Switch Penalty - single processor (10 trials) |
|  |  |

|  |  |
| --- | --- |
| Figure 3: Avg. Wait Time - multi-processor (3 trials) | Figure 4: RR Context-Switch Penalty - multi-processor (3 trials) |
|  |  |

## Analysis & Thoughts

### Round Robin

The quantum specified in the homework was simply too small to allow Round Robin to reach its full potential. The large context-switch-to-quantum ratio did not do the algorithm any favors, with the context switch penalty accounting for roughly one-fifth of the total waiting time. To make this program more effective in accordance with this project, a larger quantum value would go a long way to reduce the number of context switches and minimize wait times.

### FIFO

FIFO performed much better than Round Robin in this scenario, mostly because there were much fewer context switches with this method – only one for every process after the first. This performance, however, varied greatly and depended heavily on the ordering of the generated processes since it simply schedules the first available process without incorporating pre-emption or considering cycle lengths.

### SJF

It is easy to see from the results that the SJF algorithm produces the lowest average wait time. It is to be expected since the name is Shortest Job First, and job execution order transpires in ascending fashion, from shortest job to largest. This produces shorter waiting times, and therefore a smaller average waiting time. Its only hindrance is that it is non-pre-emptive, so a shorter job could potentially be stuck behind a much longer job if the longer job arrives first. A possible improvement upon this algorithm is the SRT, Shortest Remaining Time, that incorporates pre-emption so that a longer job can be halted in favor of a newly arriving shorter job.

### Multi-Processor

Also apparent from the data is the effect multi-processing has on average wait times. For every algorithm, implementing a multi-processor system lowers average wait time. The wait times were reduced to roughly one-fourth of the single processor arrangement in all cases. This is of course logical, since the jobs are divided evenly amongst four processors whom execute in parallel.

In addition, a small reduction in context switch penalty is afforded. Because of the job division amongst three extra processors, each of the three algorithms is guaranteed at least three less context switches overall. In the case of Round Robin, a significant reduction in context switching was observed. Combining multi-processing with a larger quantum could potentially minimize average waiting times a great deal and is the recommendation of our team that this be the strategy to shoot for when using Round Robin scheduling.

## Conclusion

To the uninitiated observer, it may appear as though Round Robin is both the best and most fair choice for process scheduling, since all jobs receive a pre-prescribed slice of time to do their work. However, due to the immense overhead observed when running a simulation with real world inhibitors such as context switch penalties, it becomes apparent that Round Robin is not always a viable solution for efficient scheduling. The obvious solution of adding more processors to the processing pool cannot be overlooked either, as this was the primary factor in receiving a low average waiting time. Though preemptive methods of scheduling might appear to be better on paper, one often encounters situations where the cycle times of each incoming process are not known. Therefore, the usefulness of non-preemptive methods such as FIFO, SJF, and Round Robin cannot be overlooked, nor overstated.