PCB (Process Control Block) Scheduler Report

Reece Harris & Calvary Fisher

Fall 2023, CS433 – Prof. Zhang

# Introduction:

For this assignment, we developed 5 classes in C++ to simulate 5 types of process scheduling algorithms: First Come First Serve, Shortest Job First, Priority, Round-Robin, and Priority Round-Robin.

# Data Structures:

For our implementation, we chose the standard library vector class due to its simplicity and efficient manipulation by index. The ability to grow and shrink dynamically proves useful since we cannot always foresee the number of processes requested at a given time. Per the template, the PCB class, and separate classes for each scheduler were used. We avoided using much of the overall parent scheduler class, as it has been a while since we worked with parent-child classes and wanted to focus on the main goal of the assignment: The scheduling algorithms themselves. In each scheduler class, we included 3 data members, as described in Table 1.

Table 1: Shared Data Members Across All Schedulers

|  |  |
| --- | --- |
| Member Name: | Description: |
| list\_pcb | Vector of PCB, representing queue of processes to run. |
| wait\_time | Vector of float, storing wait time for each PCB |
| turnaround\_time | Vector of float, storing turnaround time for each PCB |

All our schedulers include these 3 data members. We chose to store wait time and turnaround time into vectors, so a print function can quickly display results, without calculating them itself. The last slot of each vector holds the average wait or turnaround time. So, the print function does not even need to calculate averages. These two vectors have the bonus of storing process results by ID (i.e., T1, T2, T3) regardless of execution order.

The priority round robin implementation includes two additional data members, see Table2:

|  |  |
| --- | --- |
| Member Name: | Description: |
| org\_burst\_time | Vector of integers, recording original burst time for each process. |
| copy\_list | Vector of PCBs, storing all PCBs of current priority to execute. |

Storing original burst time is useful for quickly calculating wait time (see Table 3.) This only works when arrival time of all processes is 0, an assumption for this assignment.

We want to execute the highest priority processes first. Also, we do not need to preempt after the time quantum, unless we have multiple processes of the same – currently highest – priority waiting in the queue. So, for this implementation, the highest priority process is found. Then, all PCBs of this priority are removed from the queue and stored into the copy\_list vector. Round Robin scheduling is executed on the copy\_list vector, until all PCBs of that priority have run. Then, the next highest priority will be found.

Overall, these schedulers have O(N) space complexity, with N representing the number of processes to run. Most of our schedulers use roughly 3N memory with their 3 vectors. Meanwhile, priorityRR uses roughly 5N memory.

# Time complexity

## Expected vs. Actual – use larger files

We developed expected time complexity through surveying implemented scheduler functions to estimate Big O time.

For Time Complexity analysis, we focus on the three scheduler functions that actually run the scheduler: init, simulate, and print\_results. Pulling information from txt files can be considered overhead common to every scheduler.

A computer screen shot of a program code. 

Description automatically generated

We developed a simple program that produces .txt files filled with processes in the format of schedule.txt. Then, we run each scheduler through 10 000, 50 000, 100 000 and 150 000, processes. Results are graphed into a graphing calculator, and their shape is compared against expected time complexity.

### FCFS:

Overall Expected Time Complexity: O(N^2)

Explanation: More precisely, our implementation has O(N^2 + 6N) when all 3 functions run. The O(N) stems from needing to read through our vectors (6 times.) However, we have O(N^2) due to how wait time is implemented.

A computer code with text

Description automatically generated

Note the highlighted for loop. To calculate wait time, we add up the burst times of every previous process. The number of loops for j will be 1 + 2 + 3 + … + N, which per a summation formula is [N(N+1)]/2, aka N^2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scheduler: | Time for 10,000 (ms) | Time for 50,000 (ms) | Time for  100,000 (ms) | Time for 150,000 (ms) |
| FCFS | 23.503 | 5,192.06 | 28,428 | 78,313.3 |

A graph with numbers and points

Description automatically generated

Per this graph, we can see a parabolic shape, as we would expect with a O(N^2)

### SJF:

Overall Expected Time Complexity: O(N^2)

Explanation: Similar as FCFS, however, we also perform a bubble sort in init(). Overall, this produces O(N^2) as before. Notably, the something in init() seems to cause a lot of waiting, likely a bug in our sorting algorithm. Unfortunately, we did this section last, so we lacked the time to properly troubleshoot the problem.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scheduler: | Time for 10,000 (ms) | Time for 50,000 (ms) | Time for  100,000 (ms) | Time for 150,000 (ms) |
| SJF | 2,552.63 | 66,417.6 | 222,417 | 641,703 |

A graph paper with numbers and a dot

Description automatically generated

As the graph shows. SJF follows a parabolic shape (again, agreeing with O(N^2) time complexity.) However, the error within init() causes significant increases in runtime.

Due to this, further testing was not considered feasible.

# Further Testing - Overall Comparison of Scheduling Algorithms

Through our testing with the given parameter of using a time quantum of six, we have concluded that the scheduling algorithm with the shortest average times is *Shortest Job First*, with an average turn-around time of 82.5 and an average wait time of 61.25. Priority Round-Robin could be quicker with a time quantum of 5 or 1 with the given PCB set, producing a turn-around time of 61.875 and a wait time of 40.625. While a shorter time quantum may make Round-Robin algorithms quicker, it should be noted that the overhead of context switching is not calculated in this assignment. In real-world application, this factor would hinder Round-Robin, especially with a larger set of PCBs.

|  |  |  |
| --- | --- | --- |
| Scheduling Algorithm | Turn-Around Time | Wait Time |
| FCFS | 94.375 | 73.125 |
| SJF | 82.5 | 61.25 |
| Priority | 96.25 | 75 |
| Round-Robin | 143.125 | 121.875 |
| Priority Round-Robin | 106.75 | 85.5 |

# Potential Areas of Improvement:

1. The main area of improvement for this assignment would be to consolidate the ‘for loops’ that we used in the algorithms. We were not too concerned with optimizing the program and we have multiple ‘for loops’ that can probably be condensed into fewer loops to ensure that the program does not have to iterate through the number of processes over and over.
2. Additionally, there may have been a way to write the member functions for the scheduling classes in the “scheduler.h” class, since all the algorithm classes share the same member functions. Instead, we essentially duplicated the private variables for each scheduling class since we are not too familiar with how virtual functions work.

Our implementation for Priority Round Robin utilizes the additional vector of PCB’s: copy\_list to run its simulation. Hypothetically, it may be possible to develop an in-place solution instead. Our other vectors only held integers, so they’re less of a concern.

# Challenges:

1. Due to our different schedules, and full workloads, we found it challenging to meet up and work on the project together. In a couple weeks, we had worked out more consistent times to meet up.
2. We lacked a shared code space. We could discuss code and work on separate solutions. However, we could not work on the same file whenever we were not meeting. At one point, Calvary spent time building a scheduler Reece had already finished. Not an efficient usage of time.
3. Early in the project, we struggled to understand the difference between the simulate and print\_results functions. From the output and function headers alone, we were not sure which function should calculate the results.

# Lessons Learned:

1. Share study schedules and available meeting times at the beginning of the project. Communicating when we would be particularly busy (i.e., studying for a midterm) or available (i.e., class got canceled) proved helpful.
2. Establish a shared code space at the beginning of the project. Also, build a project to-do list to list remaining tasks and coordinate tasks between project members.
3. Establishing communication at the beginning of a project is invaluable (we used Discord.)
4. When confused or stuck, get help! Clarifying the difference between simulate and print\_results in office hours helped a lot. In a real-world scenario, this might be asking a manager for help, or contacting a client to understand their desired output. In this case, Prof. Zhang is like our manager and client.
5. Communicate any clarifications with the rest of the group. Notes, diagrams, etc. are helpful for keeping the group organized and effective.
6. In general, print functions should be passed whatever they need to print. They should not need to calculate that themselves. Instead, another function should calculate it, and pass it along to the print function. This boosts their flexibility (can print many times without having to calculate again.)
7. From this assignment, we gained an extensive understanding of the scheduling algorithms and how they operate. This assignment shed light on the sequence of operations taken in order to execute the scheduling algorithms. While FCFS was quite simple to implement, the other algorithms needed additional data manipulation techniques, such as bubble sort.
8. Ask for help when something is taking a while!