# Senior Project Thesis

Process scheduling - comparison and contrast

Martin Nestorov

March 22, 2019

# 1 Introduction

Every Operating System has some type of process handling capabilities, be that in the form of simple queue structure, or in some complex algorithm. This is also specific to the different types of systems that are handling the jobs. Some embedded systems do not have the capacity to handle complex operations, which forces them to have simple schedulers. One such example would be preemptive OS's, running on batch jobs.

There are several process scheduling algorithms that are used in batch, interactive, and real-time systems. These include, but are not limited to, First Come First Serve (**FCFS**), Shortest Job First (**SJF**), Priority Job First (**PJF**), **Round-Robin** Scheduling, Guaranteed Scheduling, Lottery Scheduling, **Multilevel Queue** Scheduling, etc. All of them have their advantages and weaknesses. Some are simpler and work for small systems, while others are more complex, but distribute the workload better. The purpose of this project is to analyze and compare these different algorithms, to show where they flourish and where they fall.

Another thing to consider is the type of the system that lies under the processes. In general, we can either consider a *Real-time* system, or an *Interactive* one. We are all together omitting *Batch* systems, as they are no longer viable and interesting. *Real-time* systems are such that take into consideration time as an essential goal. Typically, one or more devices can stimulate the system and it has to react accordingly in a certain amount of time. *Interactive* systems, much like the 'Real-time' ones, can and are stimulated by other programs, but don't have such a strict time constraints.

I personally find the topic of these intricate systems fascinating. All of these schedulers have something to teach us about optimization and managing complex systems. Being able to control an entire Operating Systems is no joke, so a good handle on *Process Scheduling Algorithms* is vital for any Computer Scientist. As I aspire to one day write kernel code, this project closely relates to my interests and is a good preparation for any further advancements in the fields of *System programming*. Although this project is big in size and complexity, the desire to learn proves to justify any means to achieve the goal.

> He who has a why to live can bear almost any how
>
> *Friedrich Nietzsche*

The application is created by using `C++` and the `ncurses` library. The reason to pick *C++* is due to its high granularity and fine tuning capabilities. With C++, I can control every aspect of the program with high accuracy, which is needed for scheduling algorithms. The *ncurses* library provides a nice platform for building *terminal-based* applications. It allows for easy interaction with terminals, controlling text output, color output, *IO* operations, etc. It also provides a clean and pretty *user-interface*, which is always a welcome benefit, especially if the work is done in C++. The *UI* will have several panels, indicating the different algorithms and processes that can be selected, and also what are the contents of the different `queues`. There would be information for each algorithm and statistical summaries after each round of execution. The UI aims at being both easy to use, but at the same time, be clear and helping people evaluate the different algorithms.

As a final goal, this project is to be shared and extended with anyone who is interested in learning the mechanics of process scheduling. It can be used as a learning and pedagogical tool. Future plans hold that the project be tailored for introductory courses on Operating Systems, where students will have a first hand experience of seeing how these algorithms work and what kind of outputs they produce.

# 2 Specification and Analysis of the Software Requirements

# 3 Design of the Software solution

This software uses several algorithms and data-structures that play a key role in the whole inner-workings. Because the purpose of the project is to show how different algorithms affect the process execution of *Real-Time* systems, we have to talk about each used algorithm and the accompanying data-structures.

But before we do that, we have to cover several structural decisions that have been made in order for the algorithm explanations to make sense.

Usually, in scheduling algorithms we have `PCB` blocks which hold references to the actual process, and through those blocks, we make decisions on which process should be executed next. Instead of doing this, because it adds another layer of complexity, not needed in this case, the role of a process and a PCB block is substituted with just a `process` class. It holds both the metadata found in PCBs and has the workings of processes. Thus, when in the text a reference is made to a process, a mental note should be made that it has a duality in it, for it holds two structures. The explanation of the `process` structure, as well as the other classes, can be found further down.

Another aspect that should be considered is that this project tries to imitate a *Real-Time* system. This means that the scheduler has the notion of preemptive tasks and of *IO* operations. Some of algorithms used are non-primitive and have been used in old *batch systems* and *interactive systems*. Thus, these algorithms have been tailored in order to work with a modern approach of building schedulers. When we refer to preemptive systems, it is meant that the OS decides when a process should be forced into a context-switch and when it should be taken from the `ready_queue`. Also, Real Time systems do not wait for IO to end, thus they switch to the next ready process. For instance, the **FCFS** algorithm originally was used in non-preemptive batch systems, but in this project, each process is preempted upon requesting IO operations. Then, while waiting for that process to finish with IO, the next ready one is taken from the queue.

Having these distinctions made, we can then proceed to the algorithms and data-structures used.

## 3.1 Algorithms and Data-structures

### 3.1.1 FCFS

The **F**irst **C**ome **F**irst **S**erve algorithm is one of the easiest to understand and implement. It can be looked at from many different angles. **FCFS** can be seen as a `linked-list` or a `queue`, which just serves each incoming process to the CPU for execution. When a new process is created, it is put at the back of the `ready_queue`. Then each process, one by one, is taken from the head of the queue and is given to the CPU for execution. It really depends on what type of system is running this scheduler, but in general, this algorithm, although easy, is not the most effective one to have. Because each process can take any time to finish, it can stall the whole system with its execution. For instance, if we were to have several processes and one is to be long in execution time, depending on their time of arrival, we can either quickly go through all of them, or we can wait for a long time.

In this project, the **FCFS** algorithm is created using a `vector` to hold all of the processes in sequential manner. At the start of the algorithm, we just take each next process for execution, wait for its time to live ( `ttl` ) and then we proceed to the next one in line. Because this project tries to imitate a *Real-Time* system, we also check if each process will do an *IO* operation. If so, the process is then sent to an `exec_io` routine, and the next process is then taken. This means that the `done_queue` will not be populated in the exact same order as the processes have been created in the `ready_queue`, because of these *IO* operations. Regardless, this still follows the **FCFS** pattern, and we can see that indeed, each process is taken from the head of the queue (technically vector).

**Evaluation**

**FCFS** gives us a few benefits. First, it is very simple to implement and understand. From this standpoint, it's a great algorithm for small batch systems. But on the other hand, it doesn't scale well. It can stall if there is **no** preemption and a **convoy effect** might occur, which means that all other processes wait for the currently running one to finish. To summarize:

| Pros | Cons |
|---|---|
| Easy to implement | Can be slow |
| Easy to understand | Doesn't scale |
| Works good for simple systems and batch systems | High risk of convoy effect |
| | System can stall if not preemptive |
| | Bad prediction for *Waiting time* and *Turnaround time* |

Table 1: Pros and Cons of FCFS

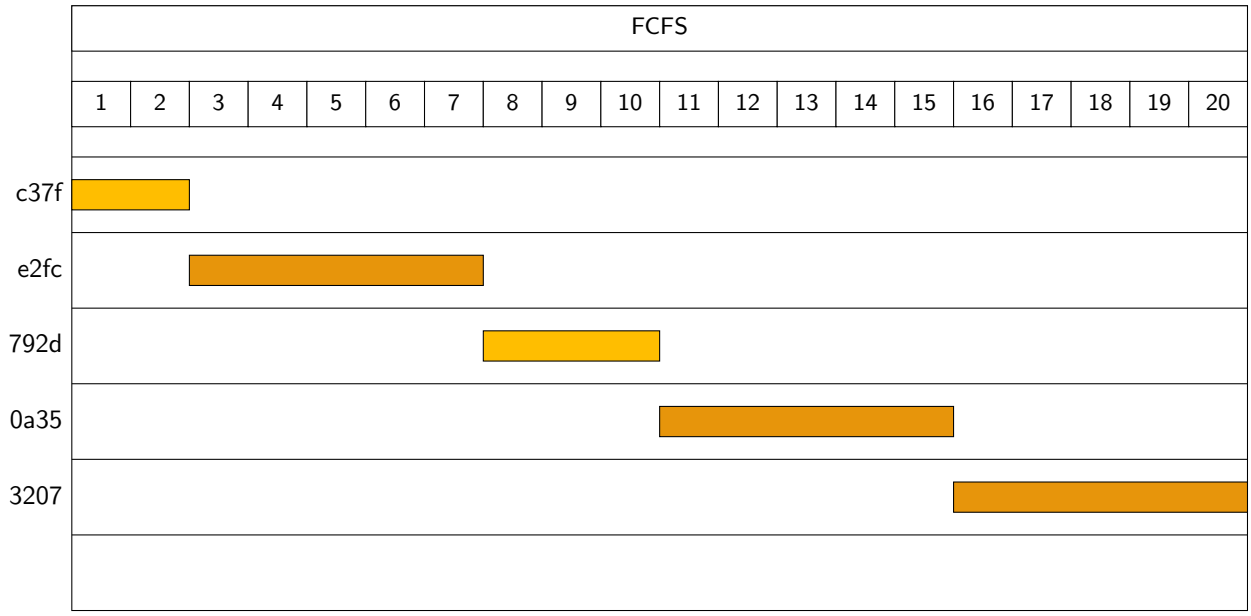
*Example*

Let's take a set of processes and see how they would hold under this algorithm. We generate 5 random processes.

| Process ID | Time To Live |
|---|---|
| c37f | 2 |
| e2fc | 5 |
| 792d | 3 |
| 0a35 | 5 |
| 3207 | 5 |

Table 2: FCFS processes table

From this, we can create the following **Gantt chart**.




Then we can calculate the average **Waiting time** and **Turnaround time**.

| Process ID | Waiting time | Turnaround time |
|---|---|---|
| c37f | 0 | 2 |
| e2fc | 2 | 5 |
| 792d | 7 | 3 |
| 0a35 | 10 | 5 |
| 3207 | 15 | 5 |
| **Average** | **6.8** | **4** |

Table 3: FCFS times table

Of course this is a fairly simple example, because the processes do not take as long to execute, and we are also not taking into consideration the fact that these processes *might* have some IO to do.

### 3.1.2 SJF

The **S**hortest **J**ob **F**irst is another easy to understand, not so easy to implement algorithm. The SJF is an optimal algorithm, because it takes the greedy approach. It tries to finish all of the shortest processes first, which would cause the whole set of processes to finish as quick as possible. But there is a downside to this. This algorithm is not practically applicable, because it either relies on information beforehand for each process, or it has to do estimations for each upcoming process. Usually there is no way to know for how long a process will execute. In general, depending on what information we have, there are two ways to approach this algorithm.

**Note:** much like the FCFS, we also take into account the fact that we might have IO operations and we also preempt every process that does such actions.

• Method 1

In a perfect world, we would be graced with the knowledge of how long each job would. Thus, one way is to look in the `ready_queue` and arrange the processes by their execution time in increasing manner (from smallest

to largest). The shortest job will run first, then the second shortest one, and so on. Because each job has a *pre-defined* execution time, which we take as a *given*, based on that value, we sort the queue. This is the simpler approach because it only relies on information that is already given to us. It is important to note that this algorithm is a special case for a **PFJ** (Priority First Job) algorithm, because we are organizing the processes based on their execution time, which would be a form of a priority measurement.

• Method 2

Continuing from *Method 1*, we do not live in a perfect world and as we mentioned earlier, because we do not know before-hand what is the *actual* time of execution for each process, we try to *guess it*. This is done by predicting the next job execution time on the basis of the previous jobs. This so called *exponential average* of the measured lengths of the previous jobs will provide a good guess as to what to expect. The *exponential average* can be defined as follows: Let $t_n$ be the length of the nth CPU burst. Let $\tau_{n+1}$ be our prediction for the next CPU burst. Then $\forall\alpha, 0 \leq \alpha \leq 1$, we define

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \tag{1}$$

Where $t_n$ is the most recent information we have, $\tau_n$ is the previous prediction, and $\alpha$ is the weight of our past predictions. If $\alpha = 0 \rightarrow \tau_{n+1} = \tau_n$, which means that the previous history does not matter. If $\alpha = 1 \rightarrow \tau_{n+1} = t_n$, which means that only the most recent history matters. A good middle value (quite literally) can be to put $\alpha = {}^1/_2$. This way, both recent and past history have equal weight on the next *exponential average*.

The expanded *exponential average* formula looks like this

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0 \tag{2}$$

Usually, $\alpha$ is less than 1, thus each successive term has less and less weight than the previous one. By using this formula, we can then make an informed decision on how to execute each incoming process.

One thing that arises in this algorithm is why we must make a guess as to which process should be executed? Why is this happening? This type of guessing is done only here, while anywhere else, we do not need to make presumptions. The reason is that for all other algorithms, we either have a constant `TIME_QUANTUM`, which is used to define each CPU burst (used in **Round-Robin Scheduling**), or we wait for the process to tell us if it's done or not (such as the case with **FCFS**). Here, however, we have to *dynamically* specify the CPU burst time on each new process execution. This forces us to keep track of each execution time.

**Evaluation**

In this case we might think that FCFS is the better algorithm, since at least it has some real life value to it, but that's not the immediate case. Although we are dealing with a hypothetical algorithm here, it is still worth it to check what are the times for SJF. If we run an example, we would see that SJF has better waiting time for each process, where the turnaround time doesn't change. Thus, SJF yields better results on average that FCFS.

| Pros | Cons |
|---|---|
| Optimal | Somewhat difficult to implement |
| A good academic exercise | Cannot be used in a real system |
| Can produce fast results | Still might have a process that stalls |

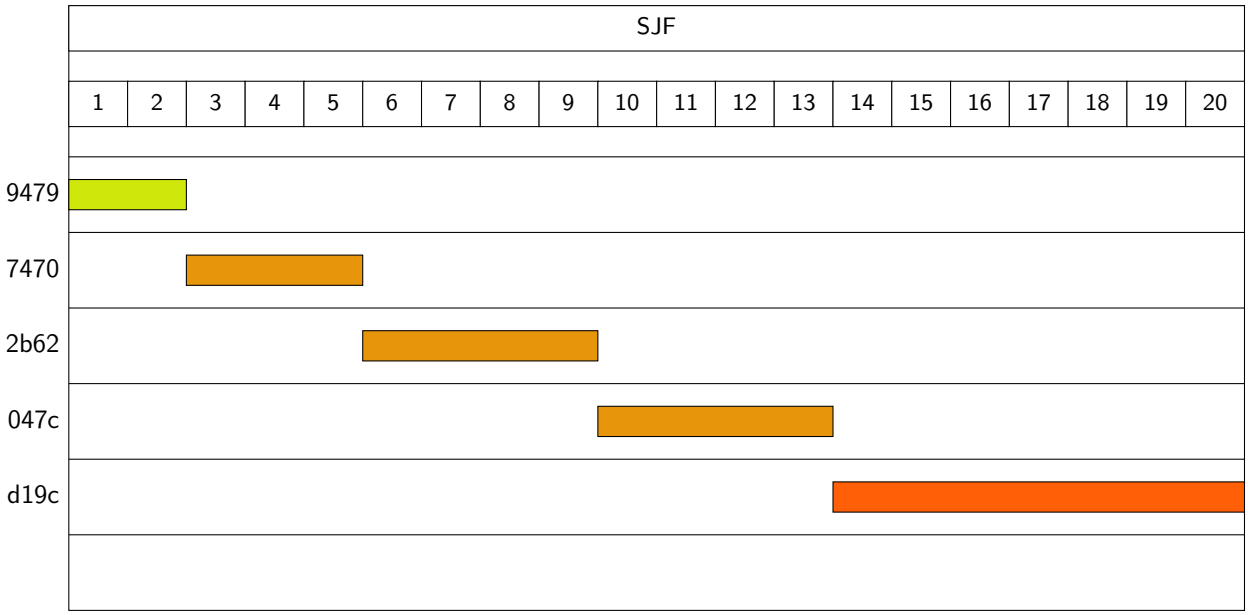
Table 4: Pros and Cons of FCFS

*Example*

Let's take again a set of processes and see how they would work. Then we can create the **Gantt chart**.

| Process ID | Time To Live |
|---|---|
| 9479 | 2 |
| 7470 | 3 |
| 2b62 | 4 |
| 047c | 4 |
| d19c | 7 |

Table 5: SJF processes table




Then we can calculate the average **Waiting time** and **Turnaround time**.

| Process ID | Waiting time | Turnaround time |
|---|---|---|
| 9479 | 0 | 2 |
| 7470 | 2 | 3 |
| 2b62 | 5 | 4 |
| 047c | 9 | 4 |
| d19c | 13 | 7 |
| **Average** | **5.8** | **4** |

Table 6: SJF times table

#### 3.1.3 Round-Robin

# 4 Implementation

This software is developed entirely with the `C++` programming language. All of the implementation uses the `C++ 11` standard (and up). The `C++ 11` standard and all of its follow-up additions, all the way up to `C++ 20`, have great benefits to creating modern, safe, and easy to manage software. Many functionalities introduced since `C++ 11` have been used to create this project. Most noticeably, the use of lambda functions, collection manipulators, threads and mutexes, and many more, play a key role.

In addition to this, the famous library `ncurses` is used in order to make working with terminal emulators easier. `ncurses` provides an `API` for manipulating the graphics and output of the terminal. Since this is an application, based on working with a terminal emulator, such a library would be of great help. The specific terminal that was used to test and run the application is `xterm`, but this was also tested on `gnome-terminal`.

The operating system used to create the software is `Arch Linux`, with additional testing environment under `Fedora 29`.

# 5 Testing

# 6 Result and Conclusion

# 7 References