Senior Project Thesis

Process scheduling - comparison and contrast

Martin Nestorov

March 21, 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 scheduling algorithms. One such example would be preemptive OSes.

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 Scheduling, Round-Robbin Scheduling, Guaranteed Scheduling, Lottery Scheduling, and Multilevel Queue Scheduling. All of them have their advantages and weaknesses. Some are simpler and work for small sequential 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 their strengths and weaknesses.

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. *Real-time* systems are such that take into consideration time as an essential role. 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 constraint.

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.

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

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.

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