Senior Project Thesis

Process scheduling - comparison and contrast

Martin Nestorov

April 27, 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. This means that the project should be able to compile on multiple machines, which are at least capable of supporting `C++ 11` and `ncurses`.

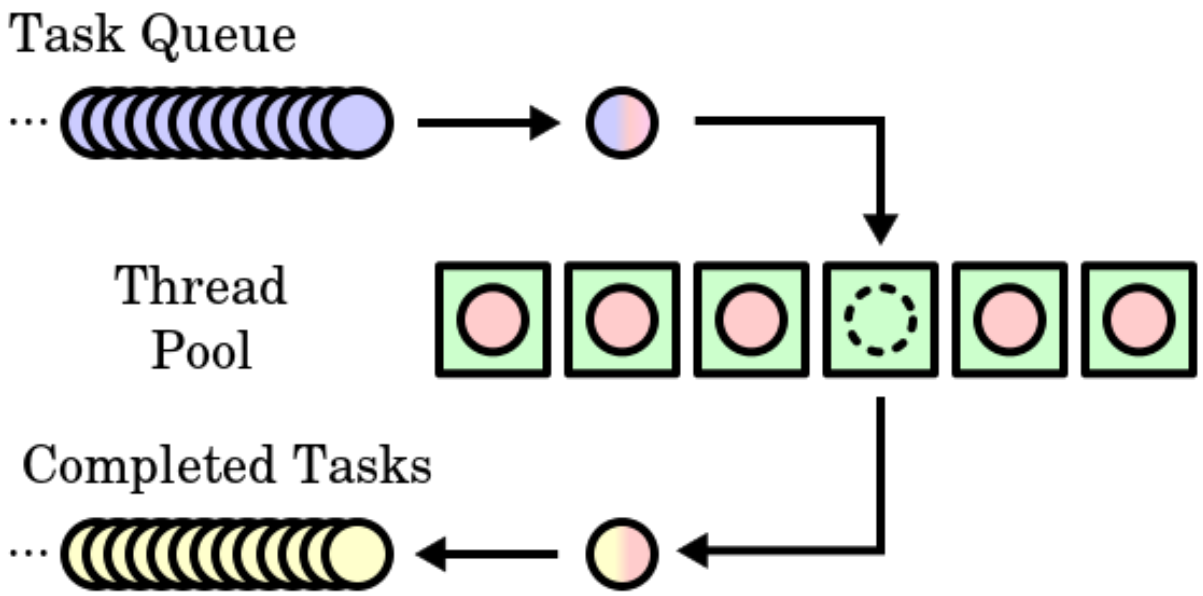


Figure 1: Thread Pool

This diagram shows us the most basic and fundamental way to treat a process scheduling system. It's just a queue with tasks, inserted into a Thread Pool, which then picks one of the many tasks (or jobs), executes them, and sends them to the Complete Queue. Rinse and repeat.

2 Specification and Analysis of the Software Requirements

2.1 Requirements

This software has a set of requirements that have to be met and covered in order to be held as working and complete, before it can be released for any pedagogical or personal usage. These specifications are separated into *functional* and *non-functional* requirements. The aim of the functional requirements is to describe what the software **should do**. Most of them aim at simplicity and ease of use of the application. The non-functional, on the other hand, try to provide a fast, extensible and stable experience to the user.

First, let's start with the analysis of the functional requirements. The software *should*:

- *Having a simple and readable user interface.* This means that all of the panels in the program should be easy to understand and differentiate between one another. Things like clear panel headlines, obvious execution patters, proper color coding of processes and commands, are a **must**. Each process should and will be assigned a priority either manually or automatically, but regardless, for each one, there should be a corresponding color that would give a general indication as to what that process's priority is. This goes hand in hand with how each process will be displayed in the different panels during execution. Because there will be commands that will create processes manually, each one should be clear, both with text and with color, what type of process is about to be created. In addition to the readability of the project, appropriate textual information should be given for each algorithm that is currently running. In the panel **Legend**, where all of this information will be held, a brief description of each action should be given, so that anyone who is executing an algorithm knows what is happening. Because this project will also be dynamically executing processes, it should also inform the user at each step what is happening. This means that the project should

have a stable state at each moment and that the user knows it. This can be achieved through either labels or headings clearly shown in color at every moment.

- *Respond to all types of inputs*. One of the more complicated software aspects of this project is the responsiveness. Each process can executed for varying times, which means that the user might wait for one second or more, and although this is the normal behavior of the application, it should be made clear to the user that this is happening. With that, the user should be restricted to the commands and inputs he/she can do at any moment. The application is build to be light and fast, but it also has to handle every response from the *UI*. This means that when an invalid character is submitted from the keyboard, the program should not be affected by it and should continue working. When a valid character is sent, the affects of that should also be made clear and responsive. Thus robustness is ensured for all users.
- *Be able to compare and analyze produced data*. Because this project has a purpose to evaluate and compare different algorithms, it should be made easy to have a set of saved *already executed* processes and what their evaluation is. Thus, for each execution, every time and algorithm is executed, it should be saved for further evaluation and should be made easily readable by anyone. Then the last 10 executions and their summaries should be put on the screen when the user enters the appropriate command and wants to see and compare them. Finally, these summaries should be saved to a text file so that they can be separately analyzed if needed. Although only the last 10 executions will be shown, all of the previously run algorithms should be saved no matter what.

And here are some of the non-functional requirements. They focus on usability, efficiency, quality, speed, etc. These things focus on **how** the software *should*:

- *Create completely random processes*. Each process can either be created with a *pre-defined* priority, or with an automatically set one (i.e. on a random principle). The processes that are randomly generated should be properly distributed and should not follow a pattern in any way. The need for such a requirement ensures that there are no hidden rules or patterns that would disrupt the evaluations and comparisons later on. That is why there has to be a proper distribution of processes with their values. For each process, the `ID` should be randomly generated to be a hexadecimal number with an even distribution between all processes, the `ttl` (time to live) should correspond to a *Gaussian Distribution* with a predefined mean and standard deviation. The same should also apply to every *IO* operation and their quantity.
- *Have high code quality*. Software craftsmanship is highly valued. It's expected that after the end of the project, maintenance and further improvements will be continually made, especially if the project is to be used for teaching purposes. If the code is not structured and prepared in such a way, that allows for further expansions and easy bug-fixing, then that would provide for a poor project life. This means that the code base should have correct comment sections, code documentation above each method, perfect separation of concerns, a valid architectural model that is followed to the end, diagrams that help understand the project and a list of future works to be done. If this is not met, then neither the original creator, nor any following contributor would want to maintain this software and any piece of code that is not regularly maintained, falls of the market and becomes useless.
- *Non-intrusive help from the User Interface*. It is very important that any form of help that comes from the system to be as inconspicuous as possible. User friendliness is achieved when the user thinks he/she have discovered something on their own. This should be done through layered error messages, that are generated on each step of processing, allowing for a decoupled, yet understandable explanation of the situation. In addition to this, all of the helper functions should have hints as to how to make the functionality of the project more approachable.

Constraints

The application should **not** try to implement every algorithm for scheduling and have a $1:1$ correspondence with real-life systems used in production. It should try to get as possible in order to do proper analysis and evaluation, without falling into the pit of meticulous pedantic, which are not worth implementing. This does not mean to

restrict the number of algorithms, or to limit it to the most simple ones, but to put a realistic boundary on what should be expected from this product. A good mixture of complexity and quantity would fit the bill perfectly.

2.2 Use cases

This project is targeted at two general use groups. One group is for those who want to test and examine scheduling algorithms, see how they work and how they present themselves in a graphical way. The other group is for people who want to learn and have a learning experience in the world of operating systems. In some sense, this project can be seen as either a research tool, or as a pedagogical tool. In both cases we see how different process scheduling algorithms work underneath the hood and see how to compare their abilities. For people who want to learn or get a better idea as to why systems are working the way they are, through running the application and looking at the source code, this project is a great place for research and learning.

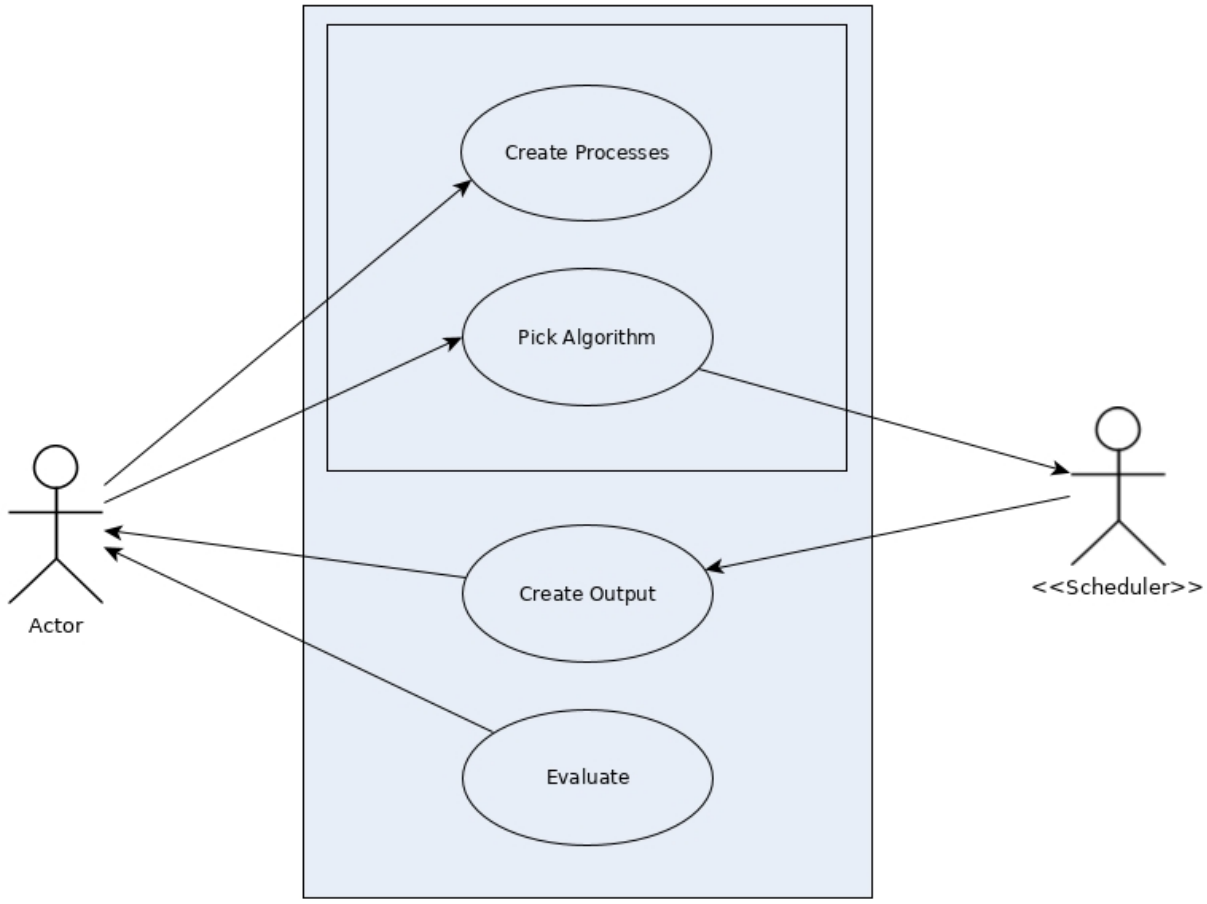


Figure 2: Use-Case Diagram

This use case diagram can help us understand how exactly to use the application. We can see that the actor in the picture would be anyone who is using the application. Because this application is not an online service, the usage is controlled solely by the user. First, the student or researcher interacts with User Interface of the app, which is part of the graphical layer. Through it, the actor is able to create different processes, where each one will be used to run in the demo algorithm. Then after the process creation is done, the actor picks the desired algorithm and then the control of the app is handed to the Scheduler component. In this diagram, we can treat

the Scheduler as another actor, because it is providing a certain type of service and is giving output in the end. After the algorithm has finished, a summary in the form of an output is given to the user. Then the user, if he/she wishes, can evaluate the received results.

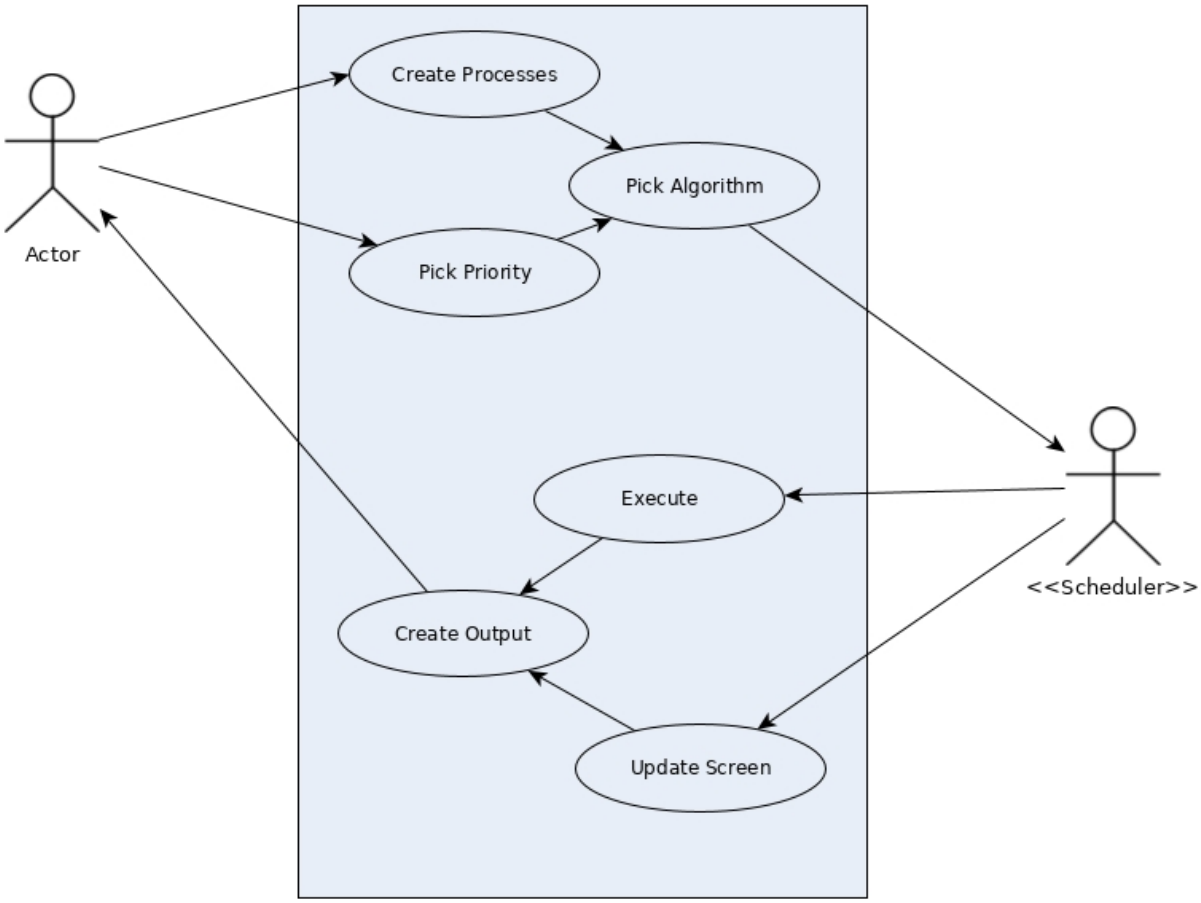


Figure 3: Narrative Diagram

In this Narrative diagram, we can see how a more detailed approach is taken to using the app.

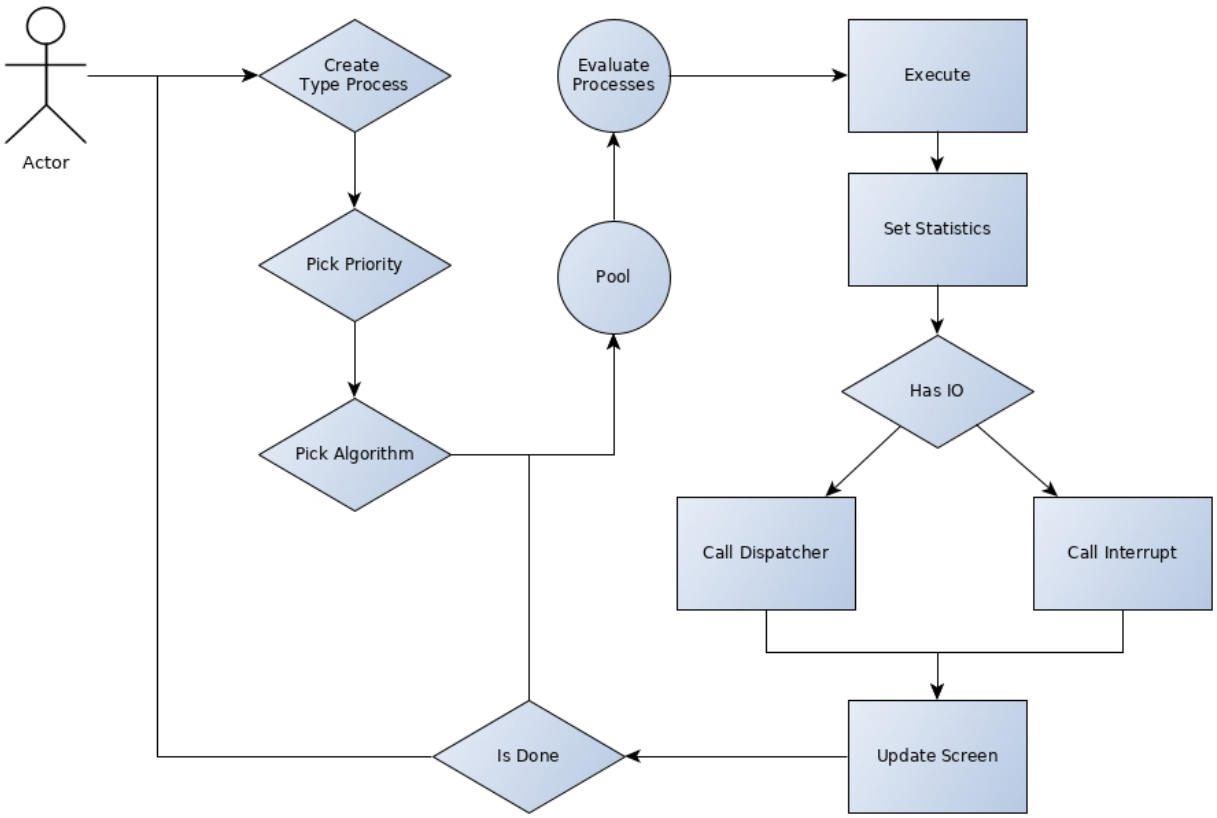


Figure 4: Pipeline Diagram

In this Activity Diagram, we can see in detail the different steps that are taken at each moment for the creating and execution of each algorithm. First, we start with a few choices that influence the processes we will see in the app. Namely, these are the priorities that each process will have. After that, an algorithm is picked. We notice here that we first have to create processes and assign them priorities, because otherwise the algorithm will be working on an empty pool of processes, which makes no sense. After that, we go to the pool entity of the program and we evaluate each processes based on the total execution time it has. If the processes, however, does have a predefined priority, we do not override it. This means that this evaluation is done for randomly created processes, and that's okay.

After this step, we then move on to the execution of the picked algorithm. The procedure here is such that we first do the "empty" work on each process and then evaluate the needed statistics for each process, then we check through a method if that process will get into an IO state or not. Based on that, we either call the dispatcher part of the procedure or the interrupt handler. If the dispatcher is called, then we do context switch and we continue with the next process. If the interrupt handler is called, then we create a separate thread that executes the IO operations.

In the end, we update the screen with all of this information and we check if the process is done. If it is not, then we just loop back to the pool and star all over until the whole pool is executed completely. If it is however, then we finish everything and give the desired output to the user!

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

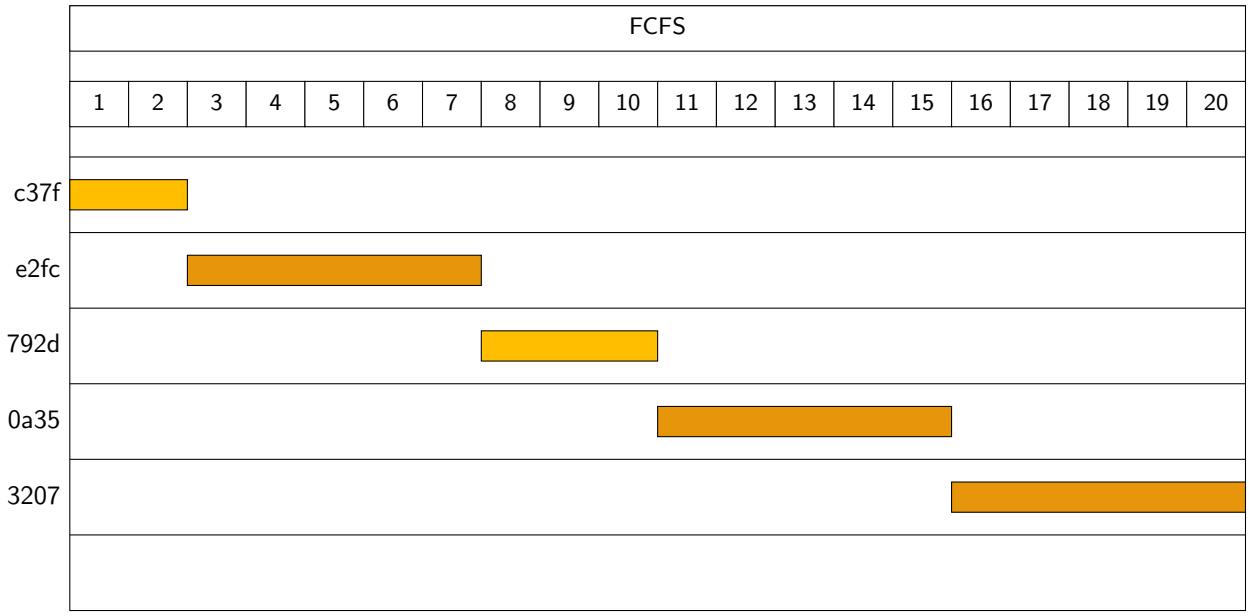
Example

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 τ_{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, τ_n is the previous prediction, and α 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, α 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

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 SJF

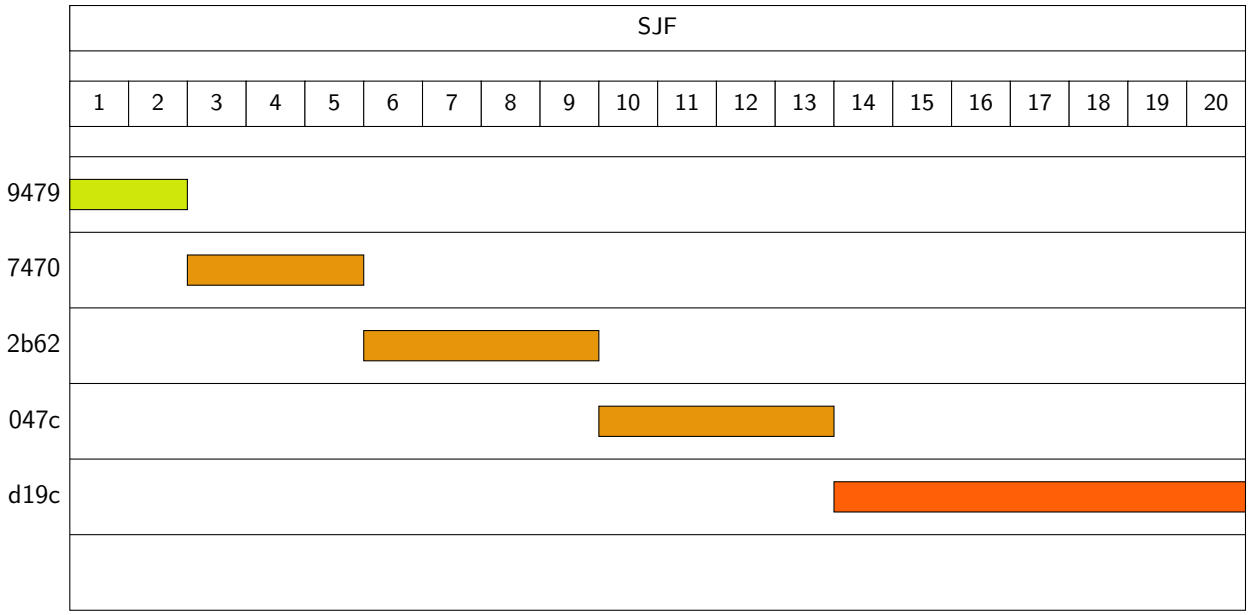
In this case we might think that FCFS is the better algorithm, since at least it has some real life value to it, but that's not the immediate case. Although we are dealing with a hypothetical algorithm here, it is still worth it to check what are the times for SJF. If we run an example, we would see 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.

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

The **R**ound **R**obin algorithm is one of the more complex, but far more efficient systems we can look at. It is an algorithm centered around the idea for having a more *fair* distribution of CPU bursts for each process. Initially used in the field of computer networking, and also following the same principle, the algorithm has a special global

variable, called a **TIME QUANTUM**. This constant with an initial value of between $10_{ms} \rightarrow 100_{ms}$, which is picked by the OS on the basis of the hardware capabilities, is used as the time each process is allowed to have in the CPU. Technically, this is the same constant that each process has as a burst. Then a rotation is done, where the next process is picked from the queue and executes with the same constant. This means that in the end, each process will execute for it's whole execution time, separated into equal chunks, each one **TIME QUANTUM** long.

Note: each of these algorithms concentrate on how long a process is executing, without trying too hard to have any other balancing factor, or to be more precise with it's predictions. But even with these small changes, we still see quite noticeable results in the final output in terms of both average waiting time and average turnaround time.

It's also good to mention that this algorithm handles IO heavy tasks great, as it is constantly in a state where it changes the running task and it can parallelize itself in a good manner. Another interesting thing is that this algorithm is used in a real modern system - the current Windows 10 operating system is using it. But because this algorithm alone cannot handle the complexity of the entire OS, it is also combined with a multilevel priority queue, in order to have a good scheduler.

Pros	Cons
Optimal	Too simplistic to be alone as a scheduler
Used in real systems	Not fair enough
Has fair scheduling	

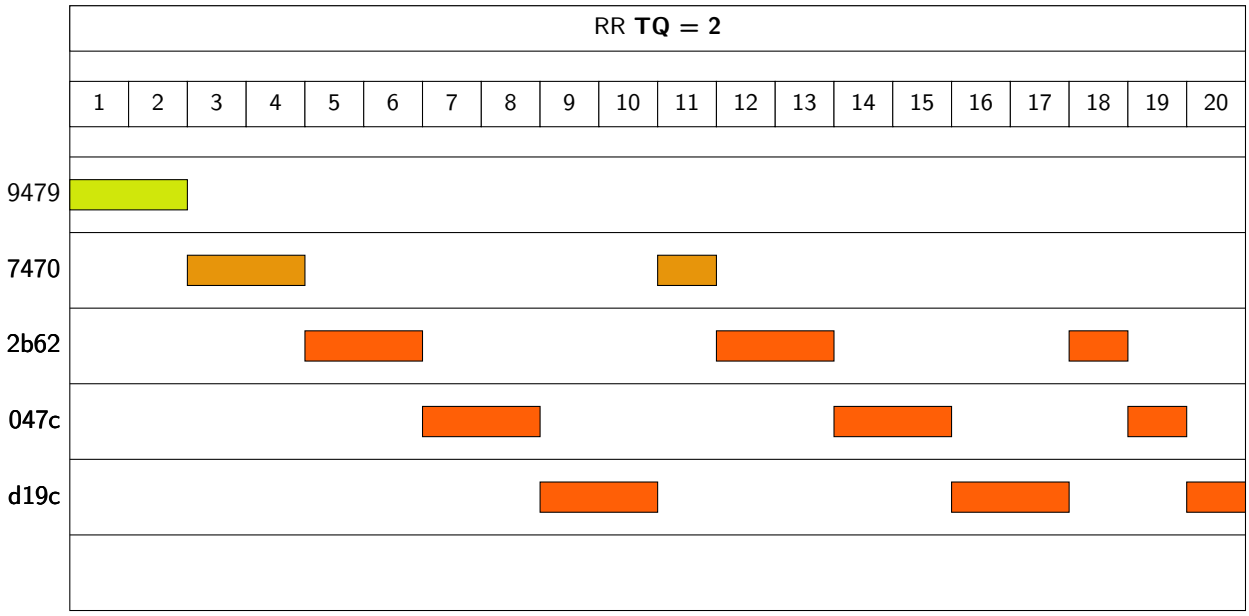
Table 7: Pros and Cons of SJF

From this table we can gather a few things. This is the first algorithm that is running in an optimal time. The complexity of the algorithm itself is in linear time, but the execution of the processes is in fact in logarithmic, meaning that we can expect optimal evaluation in the end. The reason that the algorithm is in linear time is because we are dealing with the full set of processes and we are iterating over them in a sequential manner. But the execution time of the processes is in logarithmic time, because each process will gradually go over its life time, where in the end, all of the process will almost simultaneously finish.

Let's take an example and look at a Gantt chart to see how this will look like.

Process ID	Time To Live
9479	2
7470	3
2b62	5
047c	5
d19c	5

Table 8: SJF processes table



From this chart we can see how the processes are executed in the linear fashion we mentioned. Some processes are executed exactly as the given time quantum of 2. When a process is more than the TQ, then we just execute that part of it, when it is less, we do not waste the remaining leftover time, we just take the next process immediately and execute it next. We can see that happening on the second iteration of second process and with the last 3 processes. From the chart we can also see how in the end all of the processes quickly finish one after another, which would be an indicator for the logarithmic time execution of the algorithm.

Process ID	Waiting time	Turnaround time
9479	0	2
7470	6	9
2b62	5	12
047c	5	12
d19c	5	12
Average	**4.2**	**9.4**

Table 9: RR times table

From the table we can draw the conclusion that the waiting time is significantly smaller than the rest of the processes and although the turnaround time isn't perfect, we still have a relatively fast algorithm in the end.

3.1.4 Priority Job First

3.1.5 Completely Fair Scheduling

The **C**ompletely **F**air **S**cheduler is one of the most interesting scheduling schemes we can observe. It is the famous scheduler, used by default, in the GNU/Linux Operating System. First introduced in the 2.6 patch in 2007, and shortly after optimized for the 2.6.24 release, this algorithm provides, as the name suggests, a completely fair scheduling for all of the processes.

3.1.6 A short history of Linux schedulers

Early Linux schedulers used minimal designs, obviously not focused on massive architectures with many processors or even hyperthreading. The 1.2 Linux scheduler used a circular queue for runnable task management that operated with a round-robin scheduling policy. This scheduler was efficient for adding and removing processes (with a lock to protect the structure). In short, the scheduler wasn't complex but was simple and fast.

Linux version 2.2 introduced the idea of scheduling classes, permitting scheduling policies for real-time tasks, non-preemptible tasks, and non-real-time tasks. The 2.2 scheduler also included support for symmetric multiprocessing (SMP).

The 2.4 kernel included a relatively simple scheduler that operated in O(N) time (as it iterated over every task during a scheduling event). The 2.4 scheduler divided time into epochs, and within each epoch, every task was allowed to execute up to its time slice. If a task did not use all of its time slice, then half of the remaining time slice was added to the new time slice to allow it to execute longer in the next epoch. The scheduler would simply iterate over the tasks, applying a goodness function (metric) to determine which task to execute next. Although this approach was relatively simple, it was relatively inefficient, lacked scalability, and was weak for real-time systems. It also lacked features to exploit new hardware architectures such as multi-core processors.

The early 2.6 scheduler, called the O(1) scheduler, was designed to solve many of the problems with the 2.4 scheduler—namely, the scheduler was not required to iterate the entire task list to identify the next task to schedule (resulting in its name, O(1), which meant that it was much more efficient and much more scalable). The O(1) scheduler kept track of runnable tasks in a run queue (actually, two run queues for each priority level—one for active and one for expired tasks), which meant that to identify the task to execute next, the scheduler simply needed to dequeue the next task off the specific active per-priority run queue. The O(1) scheduler was much more scalable and incorporated interactivity metrics with numerous heuristics to determine whether tasks were I/O-bound or processor-bound. But the O(1) scheduler became unwieldy in the kernel. The large mass of code needed to calculate heuristics was fundamentally difficult to manage and, for the purist, lacked algorithmic substance.

3.1.7 How CFS works

CFS tries to be "fair" to every task running in the system.

> CFS basically models an 'ideal, precise multitasking CPU' on real hardware.
>
> *Ingo Molnar, author of CFS*

Let's try to understand what "ideal, precise, multitasking CPU" means, as the CFS tries to emulate this CPU. An "ideal, precise, multitasking CPU" is a hardware CPU that can run multiple processes at the same time (in parallel), giving each process an equal share of processor power (not time, but power). If a single process is running, it would receive 100% of the processor's power. With two processes, each would have exactly 50% of the physical power (in parallel). Similarly, with four processes running, each would get precisely 25% of physical CPU power in parallel and so on. Therefore, this CPU would be "fair" to all the tasks running in the system.

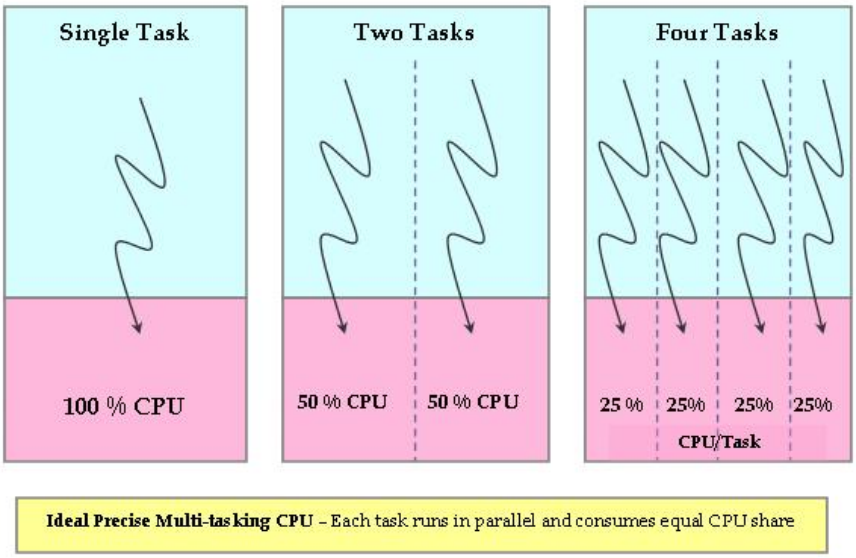


Figure 5: Ideal CPU

Obviously, this ideal CPU is nonexistent, but the CFS tries to emulate such a processor in software. On an actual real-world processor, only one task can be allocated to a CPU at a particular time. Therefore, all other tasks wait during this period. So, while the currently running task gets 100% of the CPU power, all other tasks get 0% of the CPU power. This is obviously not fair. The CFS tries to eliminate this unfairness from the system. The CFS tries to keep track of the fair share of the CPU that would have been available to each process in the system.

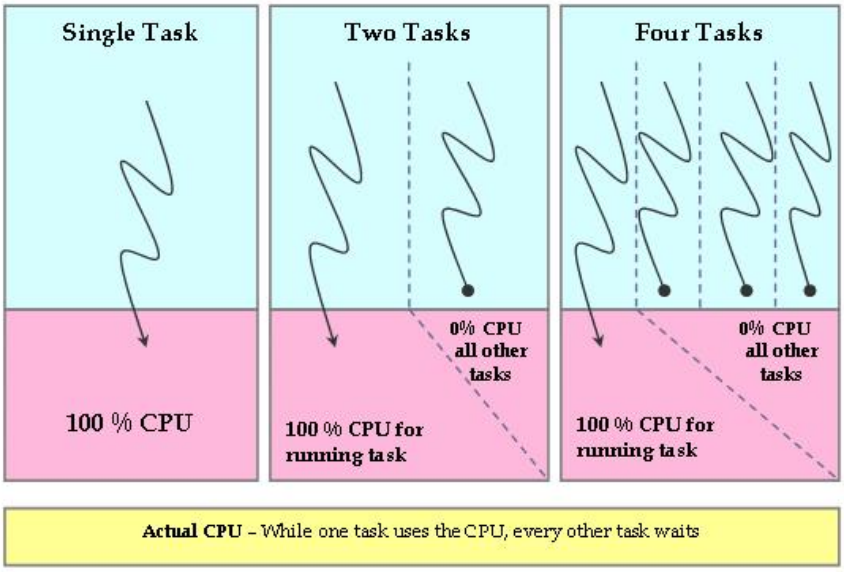


Figure 6: Actual CPU

The main idea behind the CFS is to maintain balance (fairness) in providing processor time to tasks. This means

processes should be given a fair amount of the processor. When the time for tasks is out of balance (meaning that one or more tasks are not given a fair amount of time relative to others), then those out-of-balance tasks should be given time to execute.

To determine the balance, the CFS maintains the amount of time provided to a given task in what's called the virtual runtime. The smaller a task's virtual runtime—meaning the smaller amount of time a task has been permitted access to the processor—the higher its need for the processor. The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable (for example, waiting for I/O) receive a comparable share of the processor when they eventually need it.

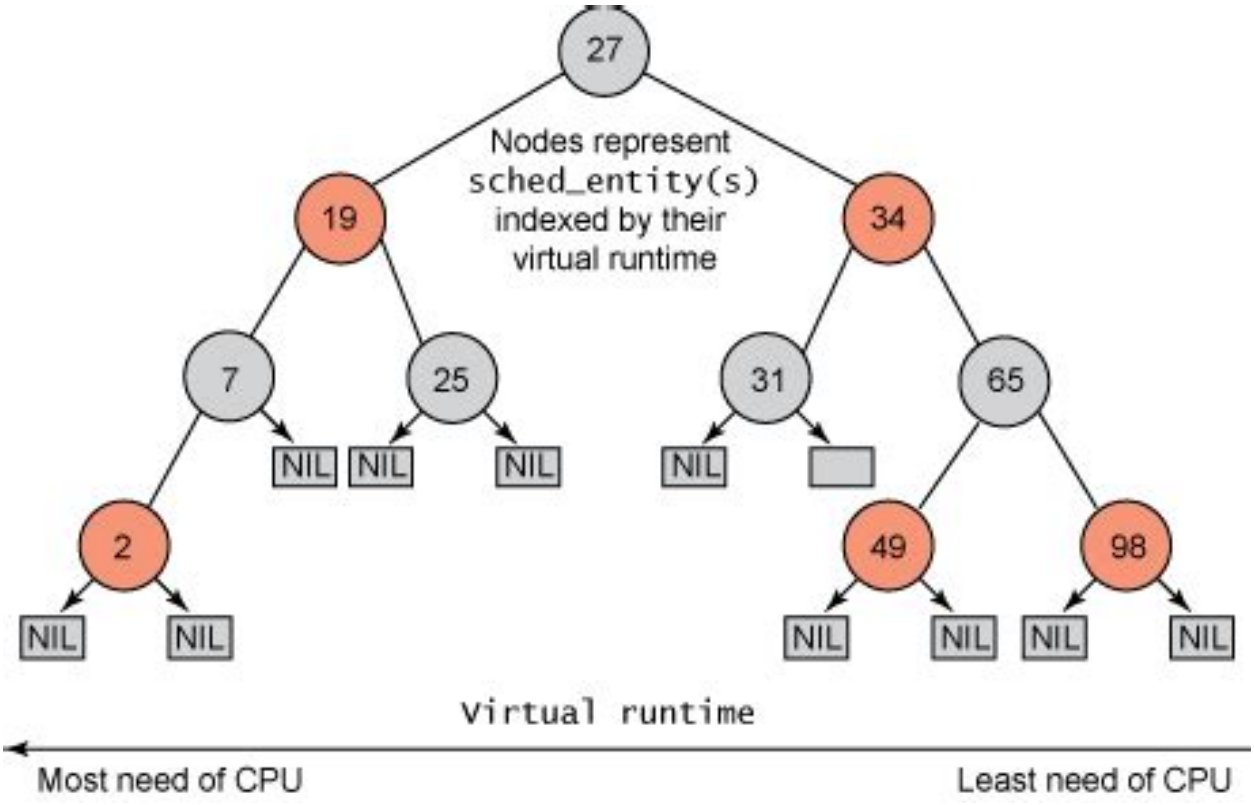


Figure 7: CFS Red-Black Tree

But rather than maintain the tasks in a run queue, as has been done in prior Linux schedulers, the CFS maintains a time-ordered red-black tree. A red-black tree is a tree with a couple of interesting and useful properties. First, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other. Second, operations on the tree occur in O(log n) time (where n is the number of nodes in the tree). This means that you can insert or delete a task quickly and efficiently.

With tasks (represented by `sched_entity` objects) stored in the time-ordered red-black tree, tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the right side of the tree. The scheduler then, to be fair, picks the left-most node of the red-black tree to schedule next to maintain fairness. The task accounts for its time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable. In this way, tasks on the left side of the tree are given time to execute, and the contents of the tree migrate from the right to the left to maintain fairness. Therefore, each runnable task chases the other to maintain a balance of execution across the set of runnable tasks.

3.1.8 Normal and Even Distributions for Processes

One very important aspect of the project is the decision on how to exactly generate processes. This seems like a strange problem, since processes are just processes, they don't even do anything in this project, but they play a key role. These jobs still have an impact on the overall performance of the program and it is not arbitrary how they are generated.

For instance, if all of the processes were the same, that would pose no variety in the global pool and thus every algorithm would have the same final output. True, each algorithm would be different, but with only one final result to show for it. This means that there must be a proper distribution made for the processes that are created, so that a real-life system is imitated as closely as possible. The question is, what type of distribution is most appropriate.

By first instinct, we would head towards a uniform distribution. Why? Because we want to treat every process as equal and reduce the algorithms being biased towards them. For instance, we don't want the **FCFS** procedure to outperform the other ones because it is having more short processes to execute, we want things to be evenly spread out. And although our intuition is heading for the right track, there is a problem. This approach is not realistic, as things in nature **do not** tend towards an even distribution. That is why we must have a good and realistic distribution in order to evaluate all of the processes properly in the end.

The underlying formula that is used for the **Gaussian Distribution** is

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{3}$$

Of course this is provided *as-is* from the standard `C++` implementations, and can be freely used for generating numbers.

But, there is a special use for a even distribution nonetheless. Namely, each process ID has to be unique and here we can create a generation algorithm that labels every process with a name that (probably) won't come twice.

Having said all of that, here is what the **Gaussian Distribution** is used for.

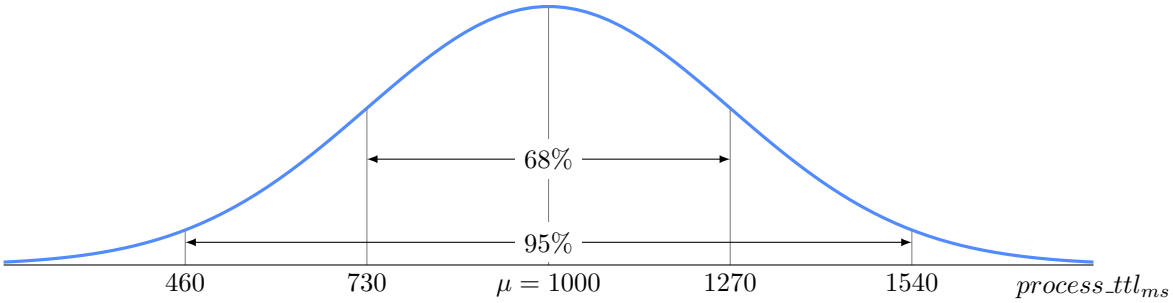
- Process Time to Live.
- Process Number of IO Operations.
- Process IO Operation duration.

From this, we can then pick the magic numbers which would go into each distribution. Because at this point, there isn't such a big deal as to how long (or short), each variable would be, a general estimate has been done. In the following table, we can see how the *Mean* and *Standard Deviation* play a role.

Type	**Mean** μ	**Standard Deviation** σ
Process TTL	1000 ms	270 ms
IO TTL	1500 ms	150 ms
Process-IO count	10	2

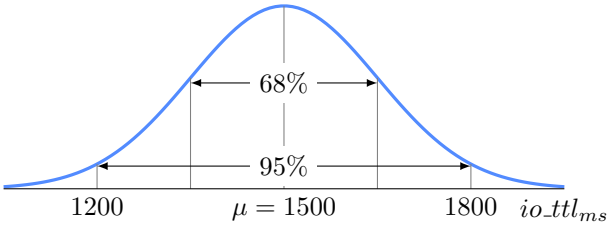
Table 10: Distributions of Process variables

From this table we can see how the processes would be created and what to expect at each algorithm run. Each new process would most probably have a `time_to_live` somewhere between 730 and 1270 milliseconds, with about 10 *IO* operations, each one taking about 1500 milliseconds to execute. Thus the Gaussian plot will be like so:

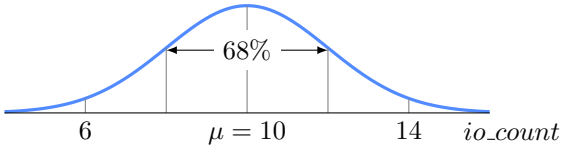


(a) Process TTL Distribution

The same would also be true for the other two variables in the table, just with different numbers.



(a) IO TTL Distribution



(b) IO count per Process

The place where we can use a **Uniform Distribution** is for the generation of IDs. Each process is named with a random *hexadecimal* value. The reason for doing this is because it simulates real processes in that each one is abstracted from what and where it came from. We care about what it does and how it is scheduled, an giving it a random name would be easiest to work with. This is done by generating a completely even number from $0 \rightarrow INT :: MAX$ and then putting that number into a converter for *hex* values. We get as an output a string that represents a unique name. The converter dose nothing more than converting that number int a hexadecimal value. The final output would look something like this.

Process ID
4b40 2e61 b88c 479c 831c d10e 9b54 bcfe a5f7 53f5 4d28
b5b2 73cd babd c920 5894 00a5 9bc8 5c6e 657c f561 ecd7
80ab 7d9b 7385 c52e 6d8a 242e 99ce fb27 0e9b f9d7 2a18
fa5b 3879 29df 2c15 9f48 3c6e 18cb cee3 d553 e15f 8dfe

Table 11: Process ID generation

Of course there randomly generated processes hold the above-mentioned distributions for *io/ttl/io-ttl*. They also have a priority, which is evaluated on each start for the different algorithms. This is shown by the colors they receive upon the start of execution.

3.1.9 Data-structures

Because we are working with a lot of different types of classes, which interact heavily during every procedure, it's a good idea to see what each object does and in what type of structure does it integrate with. The classes that are involved with the process scheduling algorithms are as follows:

- `process`

  This class holds the data for each process. Instead of using a separate `PCB` block, which points to another `process` object, the project has been simplified for ease of use, thus it incorporates both these classes into one. The `process` class keeps track of its different statistics, which are then summarized in the end for the final evaluation. Each process has *pre-defined* priority, or it can be evaluated upon creation, based on a normal distribution. Each process holds

  - *time to live* - `ttl`. The time each process it takes to execute completely.
  - *time of submission* - `tos`. The time at which the process starts executing.
  - *time of completion* - `toc`. The time at which the process finishes executing.
  - *turnaround time* - `tat`. The time between the time of submission and time of completion plus any *IO* execution.
  - *waiting time* - `wait_t`. The time a process spends in the `read_queue` and is **not** executing.
  - *priority* - `prty`. The priority each process is assigned based on its execution time, or predefined from the start.
  - *IO operations* - `ioops`. The set of IO operations each process has.
  - *process id* - `id`. The unique ID that each process has.

  In addition, this class also holds the data about its distribution, as specified in the previous section, in the for of constants.

- `pool`

  This class used as a global pool, holding all of the different processes in their different states. Thus, there are *queues* for **waiting processes**, **ready processes** and **done processes**. The pool itself is used as a general interface towards each process and its current structure. Because of this, it is much easier to perform operations on these queues, such as checking if they are empty, or clearing them, or even evaluating each process in a specific queue.

- `scheduler` The scheduler is the structure that is responsible for controlling each executing algorithm. It interacts with the pool of processes, and executes them on the specified procedure. In addition to this, the scheduler also calculates *both* the current and final **average waiting time** and **average turnaround time** of each algorithm, based on the passed processes. The class also specifies additional constants, like an `ALPHA` value for future predictions, and `TIME_QUANTUM` for equal process execution.
  The general steps of execution are like this:

Algorithm 1 Generic Scheduling

Ensure: $globals := reset()$
Ensure: $processes \leftarrow evaluated()$
while $pool \neq empty$ **do**
  $screen.update()$
  if $process \leftarrow IO$ **then**
    EXECUTE($\frac{time\_q}{2}$)
    $dispatcher :: interrupt$
  else
    EXECUTE($time\_q$)
    $dispatcher :: context\_switch$
  end if
  $awt \leftarrow calculate()$
end while
$screen.show\_summary()$

- `dispatcher`

  The dispatcher structure is used to work on the `context_switch`-es and `interrupt`s. Thus, it takes every process that has finished executing its designated time and performs either one of the two operations. When it does a `context_switch`, it goes through the procedure of $save\_state() \rightarrow restore\_state()$. When the state is saved, the current process is updated and then goes to the restore step. If the process is done, it is finalized and pushed to the `done_queue`, then the screen is updated and the reference to the next process is taken. If there is an interrupt, which occurs on any *IO*, then a separate `thread` is created and *detached* from the **parent process**. The new thread then takes the specific process and executes the next IO operation that it's waiting. After its completion, the process is returned back to the **end** of the `ready_queue`.

Having this general idea of how all these classes are constructed and what they do, we can have a clearer picture of the created data-structures that arise.

There are three vector, treated as queue, working on the basis of **FIFO**. The queues are the `ready_queue`, the `waiting_queue`, and the `done_queue`. The `ready` vector holds all of the processes that are in the `READY` state, which means that they are ready to execute their next CPU Burst(s). The `wait` vector is for processes that have interrupted the algorithm and have been sent to do IO. Generally, a process can interrupt the system because its part of its natural life span, but it can also do that when it wants to request IO and has to be sent to do that operation. In this project, only IO operations are considered to raise such interrupts, due to the fact that the rest of the operations would not matter so much to the general performance. These processes are thus labeled as being in `WAITING` state and will return to the back of the `ready_queue` once they are done with their other procedure(s). The `done` vector is the one that holds all of the processes that have *completely* finished their execution time and don't have any more work to do. These vectors are not parallel, but rather hold the same process, just in different times and quantities. The whole pool of processes is the sum of the processes found in the `ready_queue` and the `waiting_queue`. The pool is considered empty when **both** these queues do not have any process in them. This means that the `waiting_queue` can have a size of 0, while the ready one has all of the processes in it. And vise versa, the `ready_queue` can be completely empty while all of the processes are doing some sort of IO (although this situation is highly unlikely).

3.1.10 User Interface

The *UI* of this project takes an alternative approach. Because the main focus of the application is towards algorithm execution, the interface tries to be as simple as possible and to convey the idea on what is happening in the easiest manner possible. For this reason, all of the UI is created inside of a *terminal emulator*, with additional graphics added. With the help of the `ncurses` library, used for graphics in the terminal, the final product looks nice and easy to understand without being too flashy.

The structure of *UI* is as follows: 4 main panels, each one holding information for the processes and the currently executing algorithm. Let's go through them one by one:

- **ALGORITHM** panel - This panel is used to display the currently running process and its different data. Things like its priority, `ttl`, and ID, are shown at the top. Upon each algorithm completion, a summary is displayed below, indicating the *Average Waiting Time* and *Average Turnaround Time* for that algorithm. In addition to this, the name of the used algorithm together with the number of processes is displayed as well. On the very bottom of this panel, there are two miniature windows. Both of them depict the graph of the *Waiting Time* and *Turnaround Time* in terms of **amount of time** (on the $y - axis$) and number of processes (on the $x - axis$). That is, at each CPU Burst rotation, the graph shows what are the current averages. This gives an idea on how the algorithm is performing over time. When it comes to comparing, these graphs will make it easier for further evaluation and records keeping.
- **PROCESS** panel - This panel holds all of the processes that are in the `ready_queue`. When a job jumps from doing CPU Bursts to doing IO operations, it jumps out of this panel, does its thing, and comes back inside again an the end of queue. When each CPU Burst finish, all of the processes do a rotation with one to the left, where the leftmost process will be the next one to execute. If a process is done, however, then it is popped out of the panel and inserted into the next one.
- **DONE** panel - As mentioned above, this is the place where all of the done processes are held. The inserted processes are in the order of their completion.
- **LEGEND** panel - This is the final panel and it holds information for all of the commands the user can send to the program. This makes working with the application easier. On the top, the different priorities are shown, together with the key that will spawn a process with that priority. There are 7 in total, starting form 0, all the way up to priority 6. There is an additional key that creates random processes, that will not have a priority, but that will be evaluated by the algorithm at execution time. Below this, the legend also shows each of the implemented processes in the system plus the key that will start executing it. When a procedure is started, below it, a text will appear which will explain what are its features and what it does. That way, the learning and evaluation process is made easier on the user.

3.1.11 Software Architecture

The Software Architecture of the project can be seen as one of the following: *Monolithic*, *Event-driven*, or *Procedural*. If we look at the project as a *Monolithic*, we can say that it encapsulates itself into a *single-tiered* software application in which the user interface and data access code are combined into a single program from a single platform. A monolithic application is self-contained, and independent from other computing applications. The design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function. On the other hand, were we to look at it through the perspective of the *Event-driven* design, we can see how it follows a pattern promoting the production, detection, consumption of, and reaction to events. An *event* can be defined as *"a significant change in state"*. For example, when a **process** `ttl` is completely done, it changes state from `RUNNING` $\rightarrow$ `DONE`. And again, if we are to look at the project from the standpoint of a *Procedural* system, we would see that it follows a certain set of rules, which in the end, produce an output that can be interpreted and evaluated for further work. As an example, we would be able to see how the **SJF** algorithm performs under long processes.

3.1.12 Security Features

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

In the end we have to compare all this information. We would expect that in theory, some of the algorithms would be significantly slower than others.

In this table, we see how we are comparing several algorithms, based on the number of processes that are submitted to the queue.

FCFS

Num Processes	**Exec Time**$_{ms}$	**Wait Time**$_{ms}$	**Turnaround Time**$_{ms}$
20	19746	9713	1007
50	50293	24186	971
100	101362	51690	1029
200	200948	102545	1032

SJF1

Num Processes	**Exec Time**$_{ms}$	**Wait Time**$_{ms}$	**Turnaround Time**$_{ms}$
20	19814	8542	1057
50	49986	18360	959
100	99770	43420	1047
200	200805	82409	1015

SJF2

Num Processes	**Exec Time**$_{ms}$	**Wait Time**$_{ms}$	**Turnaround Time**$_{ms}$
20	73043	38126	34787
50	188321	98706	78400
100	368691	202500	159449

RR

Num Processes	**Exec Time**$_{ms}$	**Wait Time**$_{ms}$	**Turnaround Time**$_{ms}$
20	20593	17512	18125
50	50411	45391	45254
100	192607	81059	79558

7 References