

Contents

1	<u>Introduction</u>	5
2	<u>Specification and Analysis of the Software Requirements</u>	6
2.1	Requirements	6
2.2	<u>Use cases</u>	8
3	<u>Design of the Software Solution</u>	11
3.1	<u>Algorithms and Data-structures</u>	11
3.1.1	<u>FCFS</u>	11
3.1.2	<u>SJF</u>	14
3.1.3	<u>Round-Robin</u>	16
3.1.4	<u>Priority Job First</u>	19
3.1.5	<u>Completely Fair Scheduling</u>	21
3.1.6	<u>Normal and Even Distributions for Processes</u>	27
3.1.7	<u>Data-structures</u>	29
3.1.8	<u>User Interface</u>	31
3.1.9	<u>Software Architecture</u>	35
3.1.10	<u>Components</u>	37
3.1.11	<u>Deployment</u>	39
3.1.12	<u>Security Features</u>	40
4	<u>Implementation</u>	40
5	<u>Testing</u>	49
6	<u>Result and Conclusion</u>	51
7	<u>References</u>	54

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`.

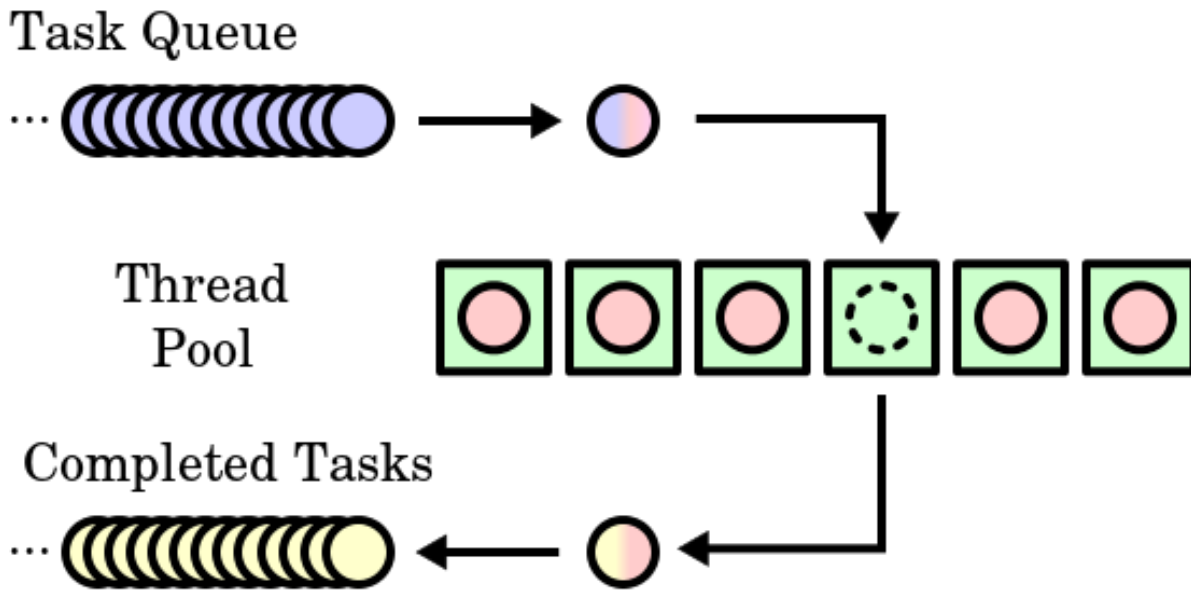


Figure 1: Thread Pool

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 patterns, 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 be 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 built 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 effects 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.

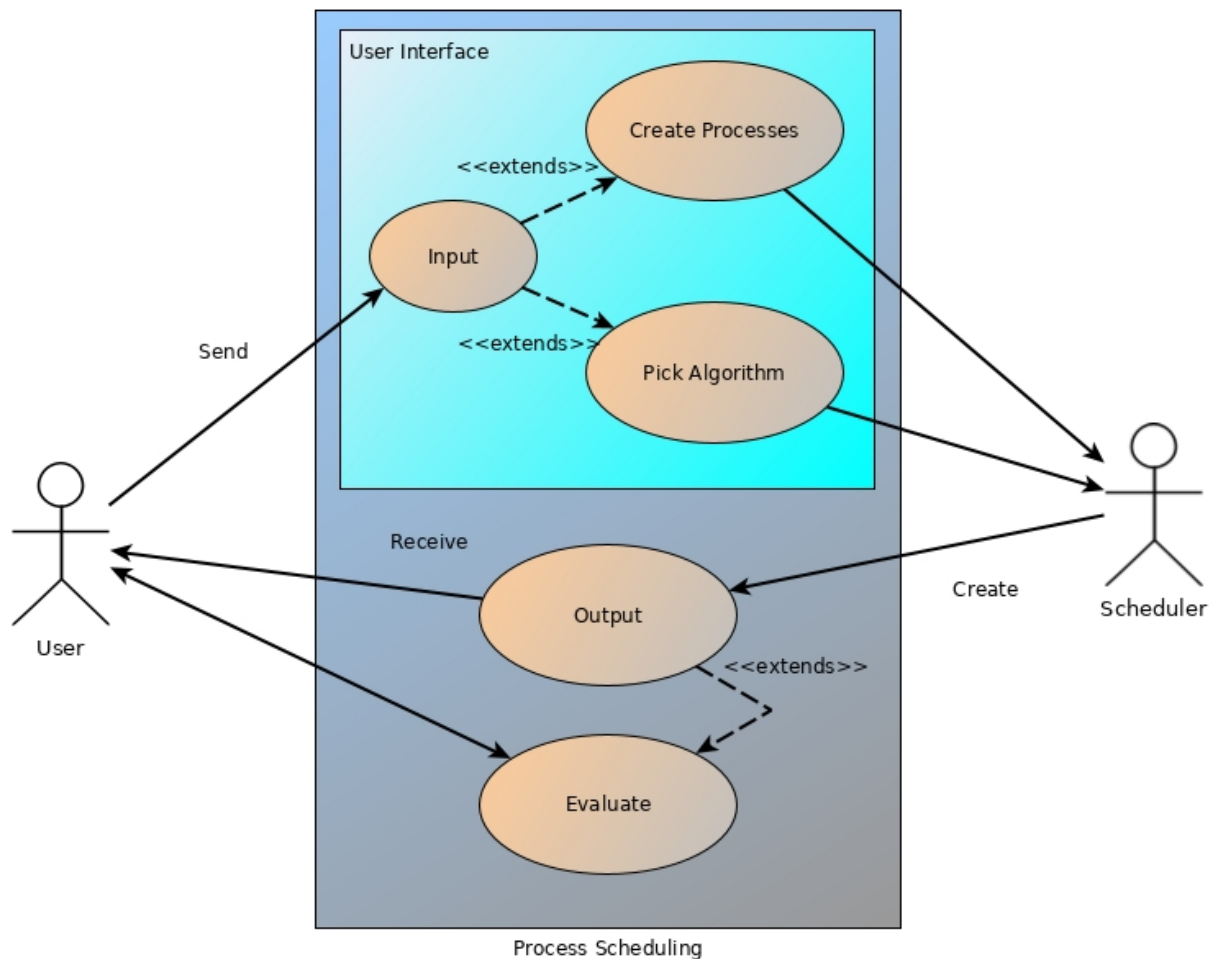


Figure 2: Use-Case Diagram

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.

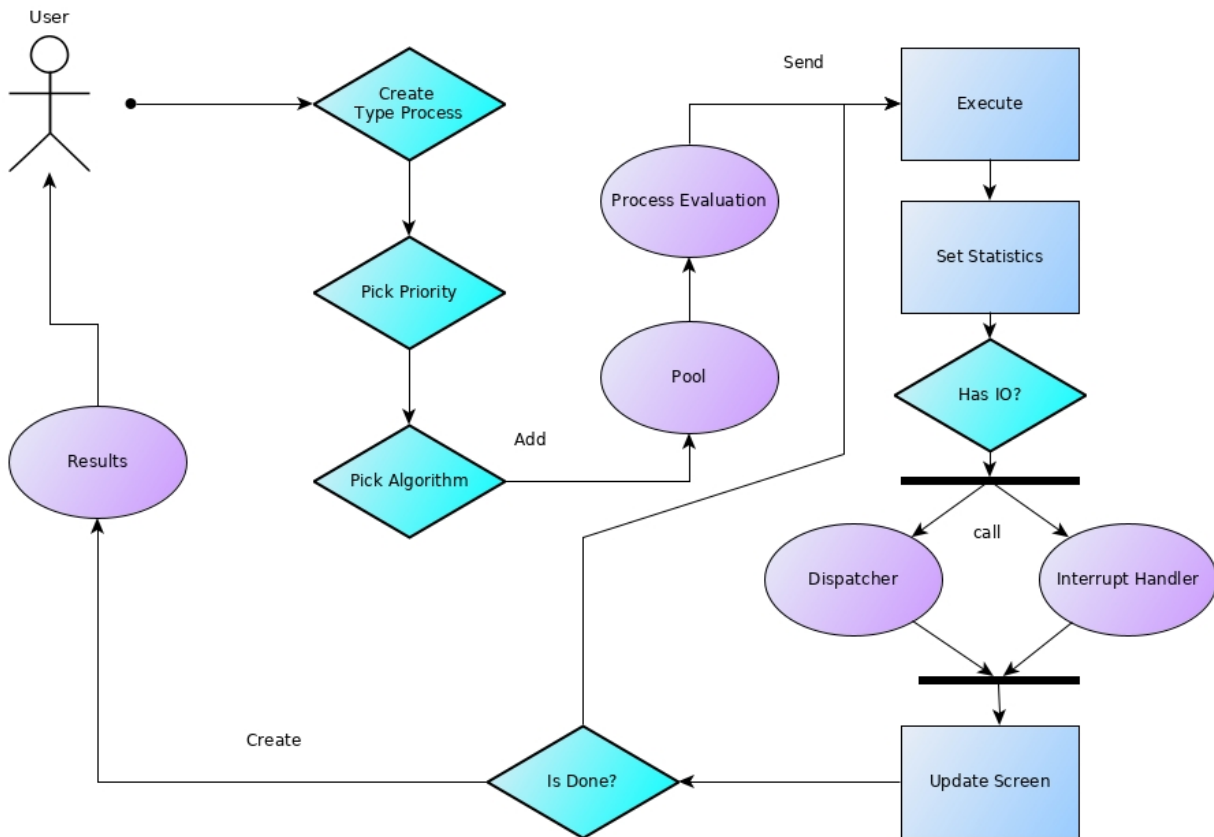


Figure 3: Activity Diagram

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 start 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!

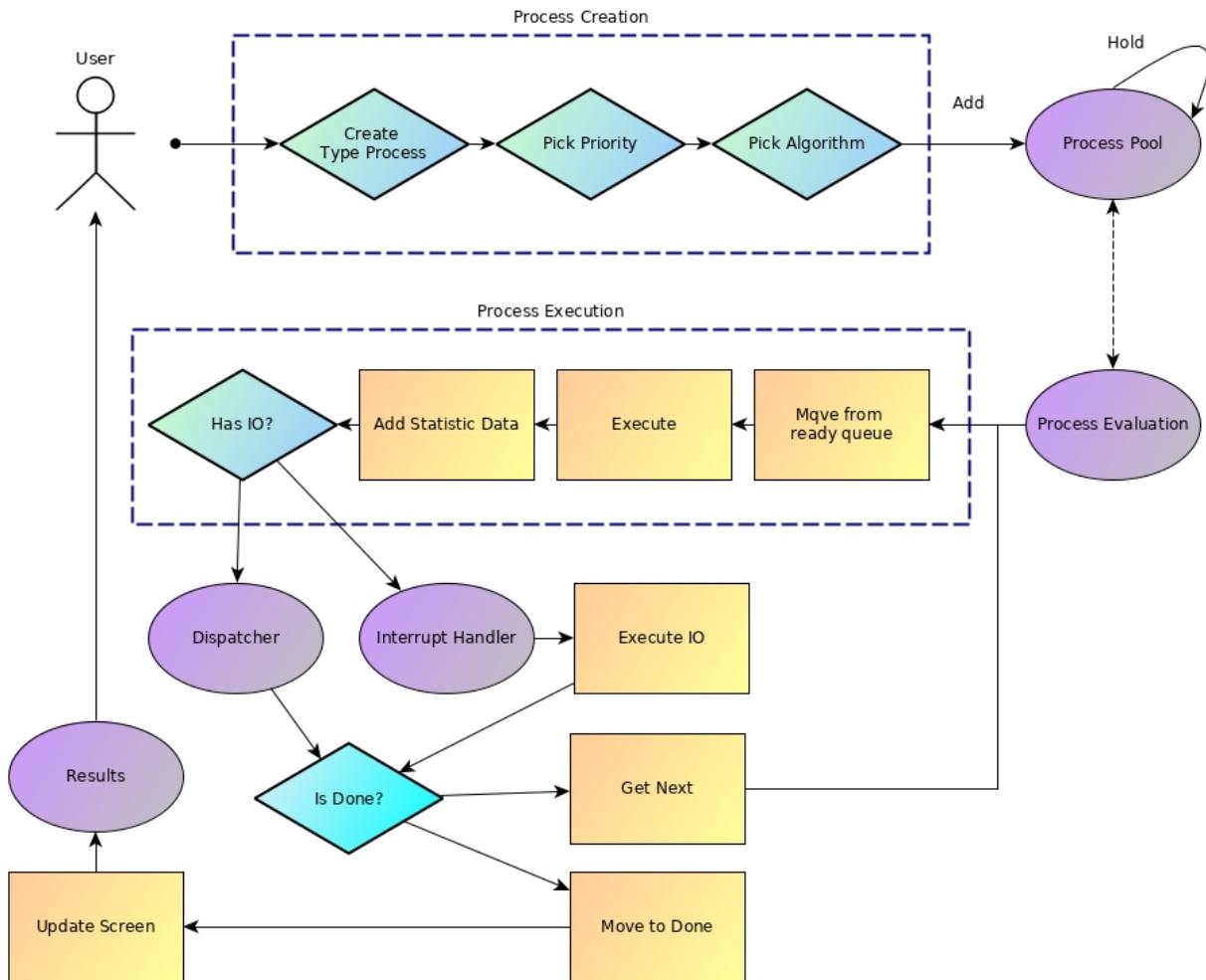


Figure 4: Pipeline Diagram

Additionally, we also have a pipeline diagram that shows us what is the general procedure of each run of the application. Through this, we can see what each part of the program does and in what context it is. First, we see that we have a process creation step, that is concerned with populating the pool of processes. For each input, we create a process and assign it a priority, either manually or automatically. Then we pick the algorithm we want to see running. All of this information is sent to the global pool of processes. We can see that the pool holds in itself all of the processes. While this is happening, we can evaluate the processes by priority, based on their execution time. We do this right before we start the main algorithm execution. When we start that part, we move each process from the ready queue into the execution step. We pick which process to move based on the selected algorithm. Then we execute the picked process for some amount of time, also dependent on the selected algorithm. When this is done, we add the statistical data that this execution has accumulated and then we check if we have to do any IO operations. Depending on that answer, we either call the Dispatcher or the Interrupt Handler. If we call the Dispatcher, then we do a context switch, otherwise, we execute the IO operation waiting

next. In the end, we check if the process is done or not. Until there are still unfinished processes, we get the next one and loop right back to the execution step. When everything is finished, we move the processes to the done queue and then update the screen and create the result, which is sent 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 **First Come First Serve** 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 <i>Waiting time</i> and <i>Turnaround time</i>

Algorithm Complexity	Execution Complexity
$O(n)$	$O(n)$

Table 1: Pros and Cons of FCFS

From this table we can also see how the complexity of the algorithm together with its execution complexity align. The reason that FCFS is in linear time when it comes to the algorithm complexity is because we are iterating over the whole bunch of processes in the pool. We are doing that in a sequential manner, meaning that we would not be skipping over anything. This is fine, but when we look at the execution complexity, we can see that we also have linear time. This is not the most optimal complexity we can have, because we are going through every process and we are also blocking the CPU, if that process has a long execution.

Example

Let's do an example in a set of a small group of processes, which would give us a more visual idea of how this algorithm might perform. In a set of 5 randomly generated processes, with decreased execution time (in order to make calculation easier), we have the following table.

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

Table 2: FCFS processes table

We notice that the processes are not ordered and that they do not have any priority assigned to them yet. This is as expected, because we want the algorithm to make that choice automatically. We also assume that the

submission time of each process is in the manner in which they appear. Now that we have this data, we can put it in a Gantt chart to see how they execute.

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

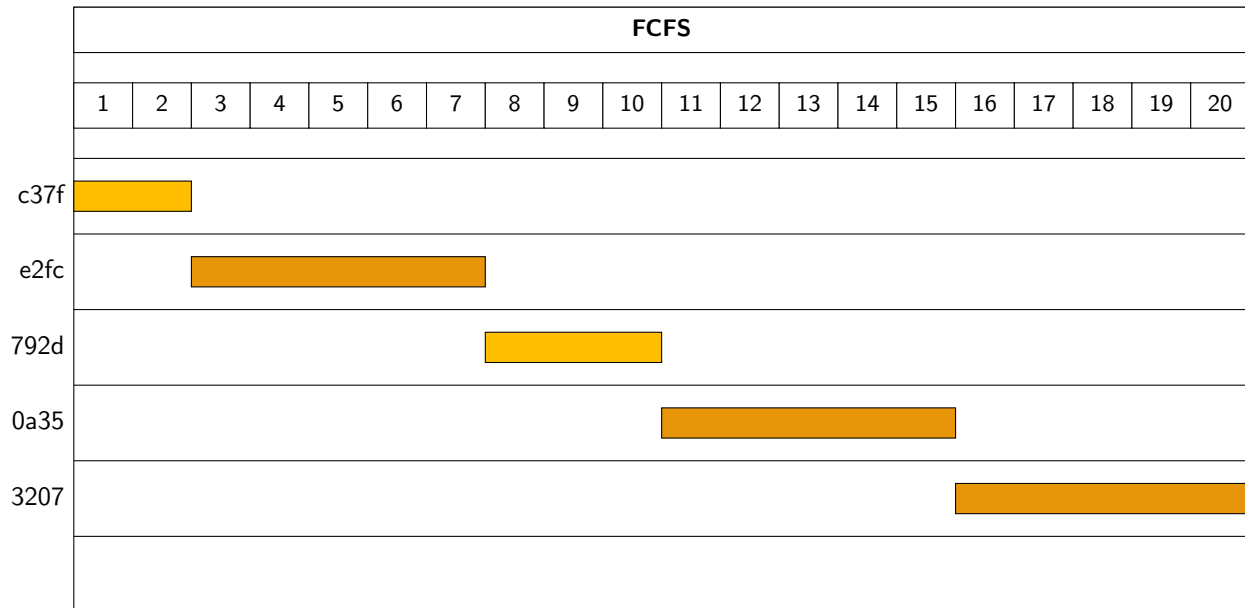


Figure 5: FCFS Gantt Chart

From the Gantt chart we see a few things off the bat. First, the processes have been evaluated in terms of priority and have been colored in their respective color. Then, we see that the time of submission is the time in which they start execution for the very first time. We can also see that they are exactly in the order listed in the table above, which is the expected behavior. Then, if we calculate and average all of the data in order to get the **Waiting Time** and the **Turnaround Time** for the whole batch, we would get the following table.

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

From this, we can see that the waiting time is quite large, even for such short processes. The waiting time is longer than the average process execution, this means that most of the time, processes are waiting in the ready queue and are not working. The turnaround time is short. In fact, for each process, the turnaround time is the time it takes for the process to execute. This is to be expected, because this is the time between the time of submission and the time of completion for each process. And although it is short, it doesn't mean that the whole algorithm did all that good. For instance, if we are to take the longest processes first and the shortest ones last, then we would have even larger waiting time, making the whole execution much slower.

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 **Shortest Job First** 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 n th CPU burst. Let τ_{n+1} be our prediction for the next CPU burst. Then $\forall \alpha, 0 \leq \alpha \leq 1$, we define

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

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

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

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

Algorithm Complexity	Execution Complexity
$O(n)$	$O(n)$ - without prediction $O(n)$ - with prediction

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 than FCFS.

From a complexity perspective, we can see that this algorithm is also running in linear time and also execution in the same time. But because we are also considering the case that we can implement such an algorithm with some prediction calculations, then the algorithm for execution becomes a little bit better and that we can expect a bit more performance boosts. The complexity in theory is still the same, but in practice, we would see better results.

Example

As before, we want to visualize how this works. With a new set of processes and with now execution times, we have the following table. Note that we also take into account the previous assumptions about the time of submission and the priority.

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

Table 5: SJF processes table

With these numbers, we would want to enforce the notion that there are some processes that are marginally slower than the rest, such as the last one. In comparison, it is 3 times slower than the shortest job. This is what

we want, because now we can compare an even more unfavorable case with the previous algorithm and see how this does.

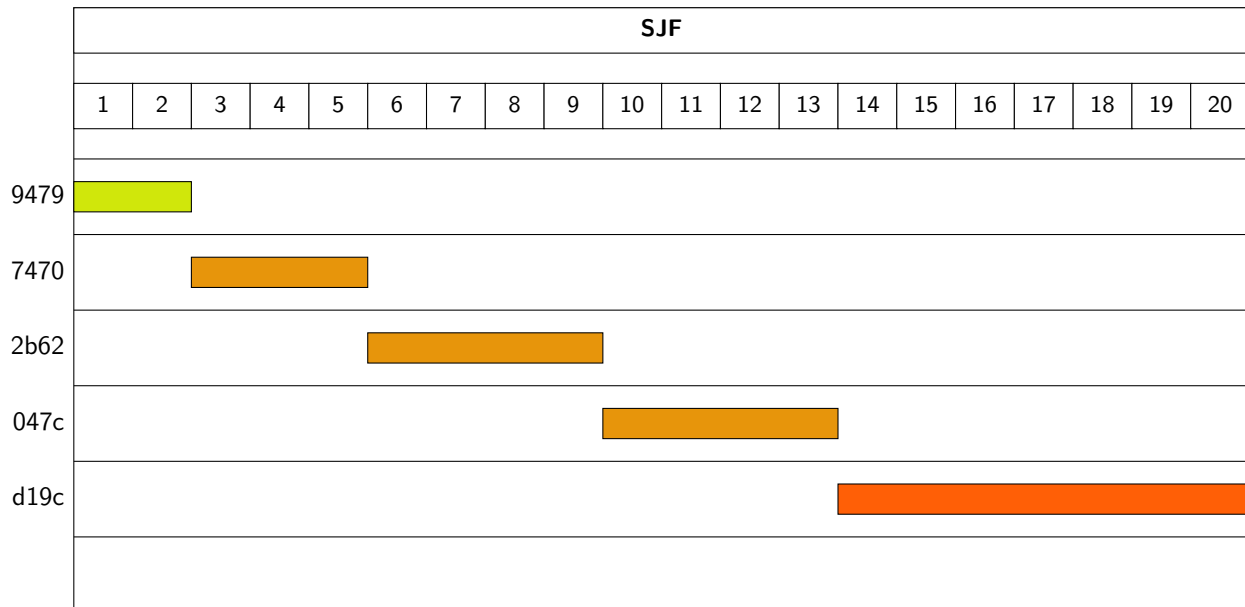


Figure 6: SJF Gantt Chart

We see that the algorithm does its job and evaluates each process accordingly and then it also sorts the processes based on their priority. This can be seen by the color arrangement each one has. The longest one is at the end and is executing for the longest time, while the shortest process is first and is the easiest to execute.

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

The results are interesting. First we see that although the more complicated set of processes, we get even less waiting time in the end. This means that with the same time and algorithm complexity, we are getting better results from this SJF algorithm. We can also see that the turnaround time is the same, which is expected, because we are not changing the state of execution for the processes and they are still executing for their complete time.

3.1.3 Round-Robin

The **Round Robin** 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 its 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 its 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.

Evaluation

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

Algorithm Complexity	Execution Complexity
$O(n)$	$O(\log(n))$

Table 7: Pros and Cons of RR

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.

Example

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

Process ID	Time To Live
6c8b	2
33e9	3
7ff7	5
d3e4	5
cce5	5

Table 8: RR processes table

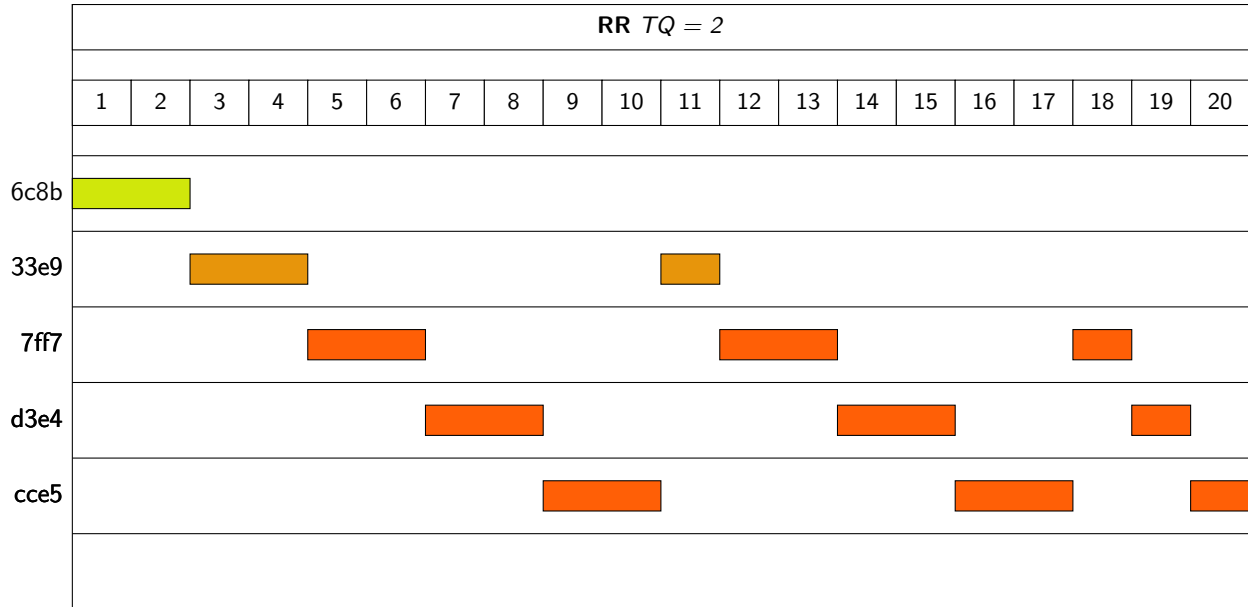


Figure 7: RR Gantt Chart

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
6c8b	0	2
33e9	6	9
7ff7	5	12
d3e4	5	12
cce5	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

The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority(p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. Note that we discuss scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this application, it has been chosen that there would be 7 priorities, starting from 0, all the way to 6, where the highest priority is 6 and the lowest one is 0.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the Operating System, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, factors. Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run, or the computer system will eventually crash and lose all unfinished low-priority processes. With this application, we cannot have such a problem, because of the simple case that there are no new processes that are being entered during algorithm execution. Although the problem can easily be simulated and reproduced, changing the whole system just for that demonstrations seems unnecessary.

Evaluation

Pros	Cons
Used in real system	Has starvation problem
Can be optimized to good speeds	Needs additional system to work good
Allows for a more interactive system	Not completely optimal

Algorithm Complexity	Execution Complexity
$O(n)$	$O(n)$

Table 10: Pros and Cons of PJF

The Priority Job First algorithm is one that is a key component to other algorithms, but on its own, it cannot outperform other algorithms. It is used in conjunction with other implementations, in order to build more stable systems, but standalone it cannot fair all that good. It does allow for a more interactive system, where the user can influence the process priority, but on the other side, it does have a serious problems with starvation on processes. There are fixes to this problem, but most of the time, they are quite complicated.

Example

Process ID	Time To Live	Priority
cd23	10	3
12a0	1	1
ff83	2	4
43bb	2	5
9b9a	5	2

Table 11: PJF processes table

In this table for the example, we have one additional filed, which is the specified priority. In this specific case, we take it that the lower priority number means that the process is more important. That is why we would consider that the second process is with the most importance, while the second to last one is the least significant one. Having this in mind, we can plot the execution of the processes and see where they fair in the Gantt chart. We would be expecting that the results not to be better than the previous algorithms.

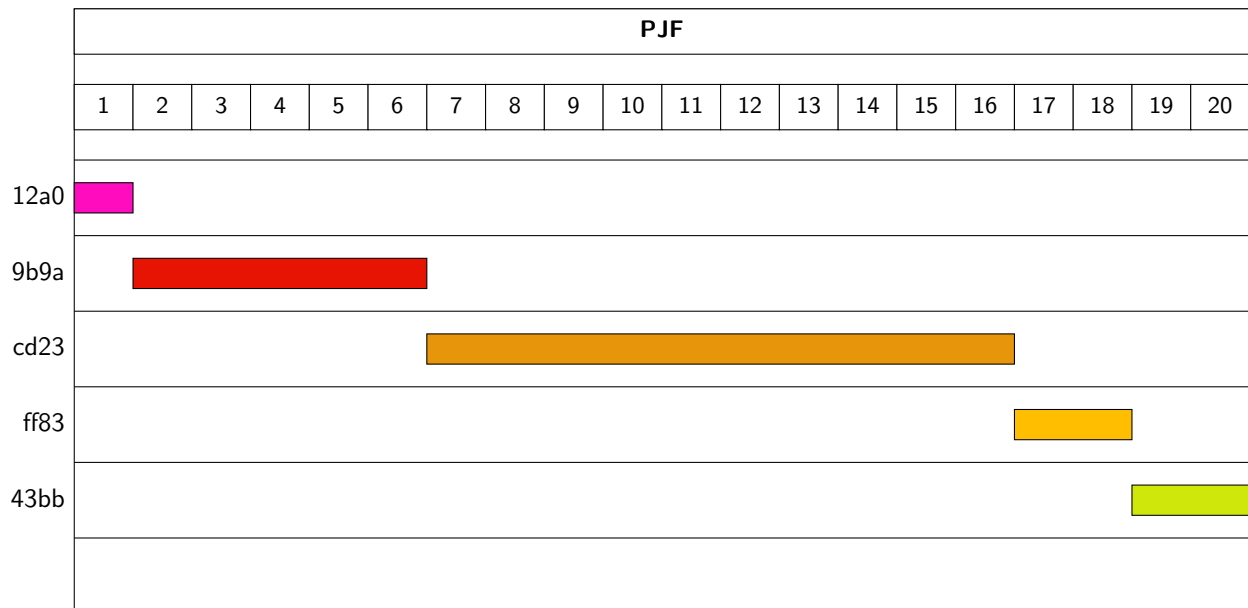


Figure 8: PJF Gantt Chart

The Gantt chart shows us that the processes are executed in a sequential manner, based on their predefined priority. As they are colored, we can see that the process with the highest priority, regardless of the fact that it doesn't have the best execution time, is scheduled first. After that the rest follow on the same principle. If the

user has specified that this is the order of execution he/she want the processes to be in, then this is fine, but from an evaluation stand point, we can see that this doesn't fair well.

Process ID	Waiting time	Turnaround time
cd23	6	10
12a0	0	1
ff83	16	2
43bb	18	2
9b9a	1	5
Average	8.2	4

Table 12: RR times table

The average waiting time is way too high for our liking and as we would expect, the turnaround time is still the same. It is clear that this algorithm isn't all that good alone and it needs another structure. Usually we would combine this with a multilevel queue, which would create a multilevel priority queue. This is what Windows 10 uses as a scheduler.

3.1.5 Completely Fair Scheduling

The **Completely Fair Scheduler** 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.

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.

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.

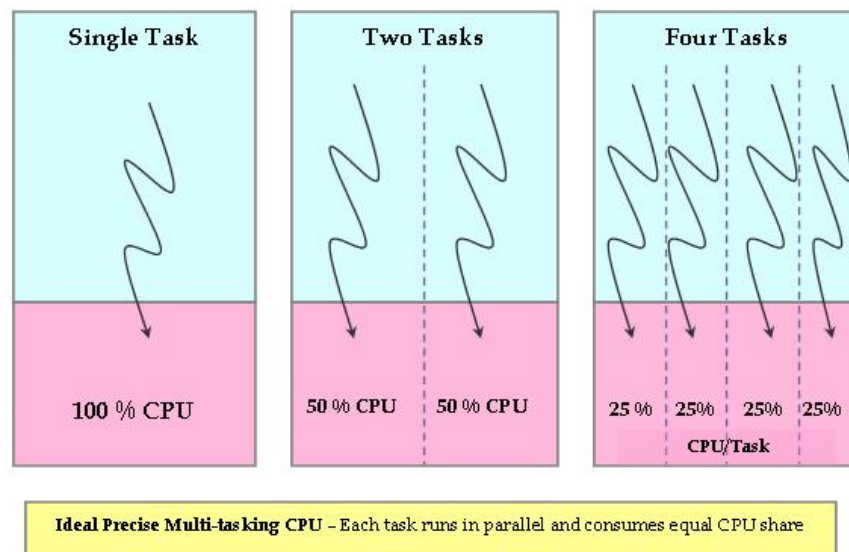


Figure 9: Ideal CPU

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.

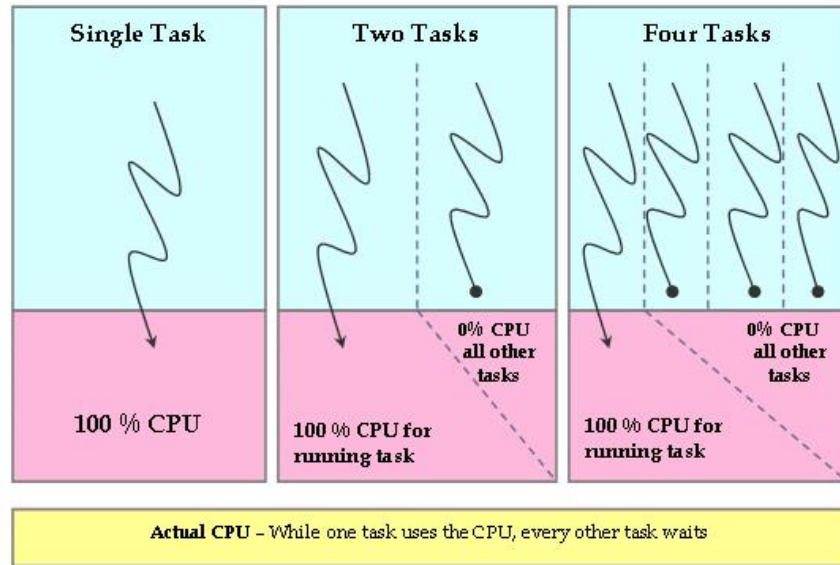


Figure 10: Actual CPU

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.

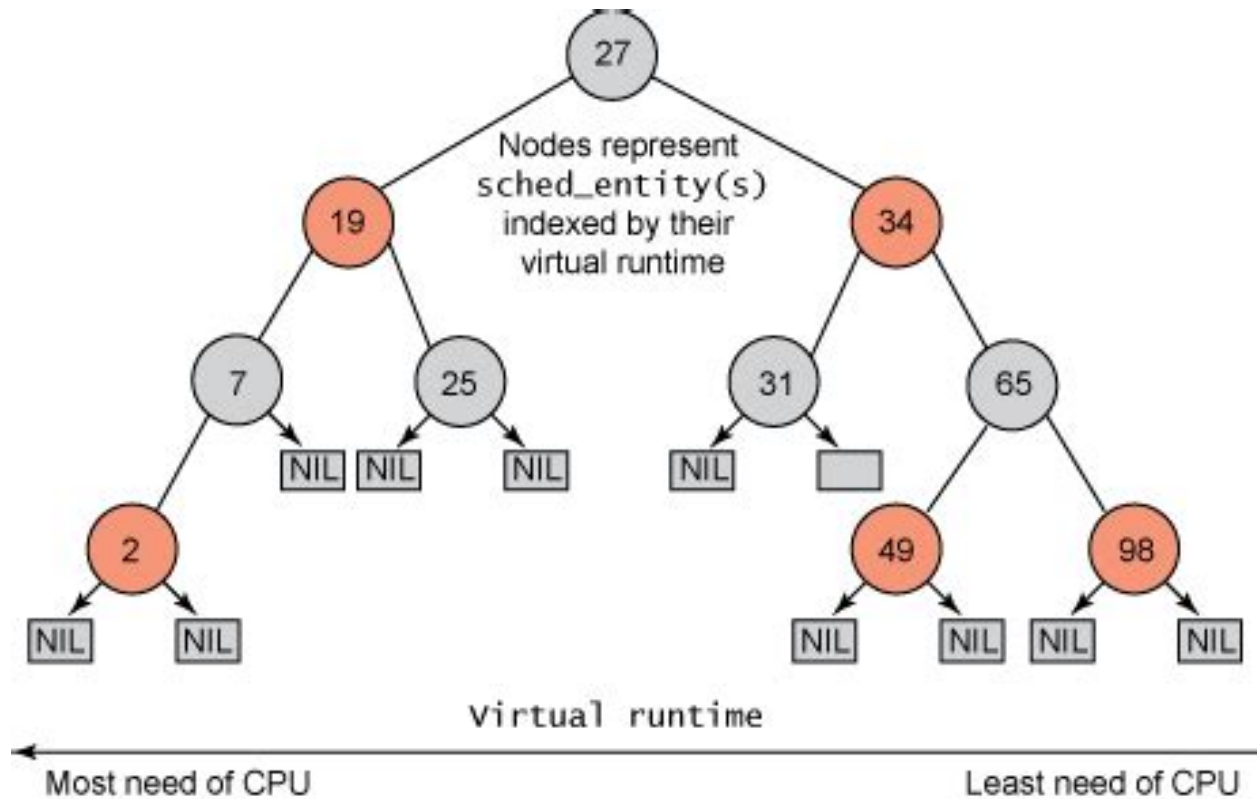


Figure 11: CFS Red-Black Tree

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.

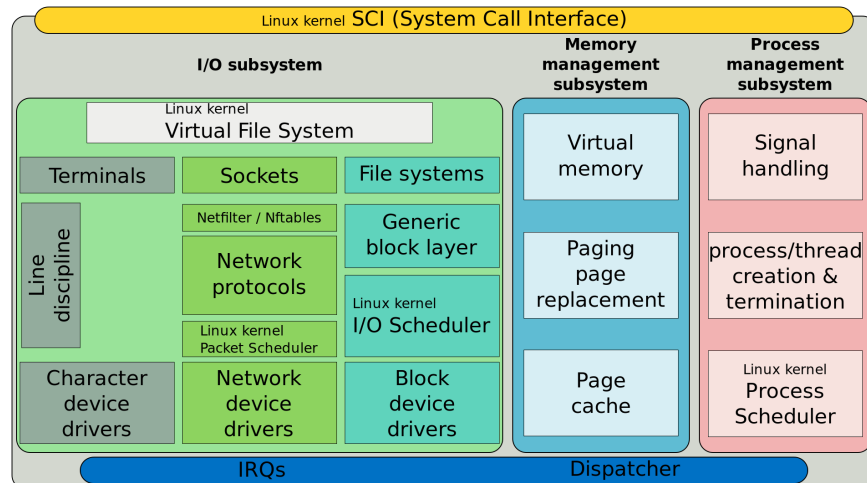


Figure 12: Linux Kernel Simplified

Here we see a simplified version of the kernel where the scheduling system can be seen in action and how it interacts with the rest of the system. The dispatcher is in a close relationship with the IRQs and has a wide reach to all systems.

A "maximum execution time" is also calculated for each process. This time is based upon the idea that an "ideal processor" would equally share processing power amongst all processes. Thus, the maximum execution time is the time the process has been waiting to run, divided by the total number of processes, or in other words, the maximum execution time is the time the process would have expected to run on an "ideal processor".

When the scheduler is invoked to run a new process, the operation of the scheduler is as follows:

- The leftmost node of the scheduling tree is chosen (as it will have the lowest spent execution time), and sent for execution.
- If the process simply completes execution, it is removed from the system and scheduling tree.
- If the process reaches its maximum execution time or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spent execution time.
- The new leftmost node will then be selected from the tree, repeating the iteration.

If the process spends a lot of its time sleeping, then its spent time value is low and it automatically gets the priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running.

CFS is an implementation of a well-studied, classic scheduling algorithm called weighted fair queuing.

Originally invented for packet networks, fair queuing had been previously applied to CPU scheduling under the name stride scheduling. However, CFS uses terminology different from that normally applied to fair queuing. "Service error" (the amount by which a process's obtained CPU share differs from its expected CPU share) is called "wait_runtime" in Linux's implementation, and "queue virtual time" (QVT) is called "fair_clock".

The fair queuing CFS scheduler has a scheduling complexity of $O(\log N)$, where N is the number of tasks in the runqueue. Choosing a task can be done in constant time, but reinserting a task after it has run requires $O(\log N)$ operations, because the runqueue is implemented as a red-black tree.

CFS is the first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.

What is a Red-Black Tree

The real beauty in this algorithm is the fact that it uses the famous Red-Black tree which is a self-balancing tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

The balancing of the tree is not perfect, but it is good enough to allow it to guarantee searching in $O(\log n)$ time, where n is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in $O(\log n)$ time.

Tracking the color of each node requires only 1 bit of information per node because there are only two colors. The tree does not contain any other data specific to its being a red-black tree so its memory footprint is almost identical to a classic (uncolored) binary search tree. In many cases, the additional bit of information can be stored at no additional memory cost.

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red-black tree:

- Each node is either red or black.
- The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
- All leaves (NIL) are black.
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

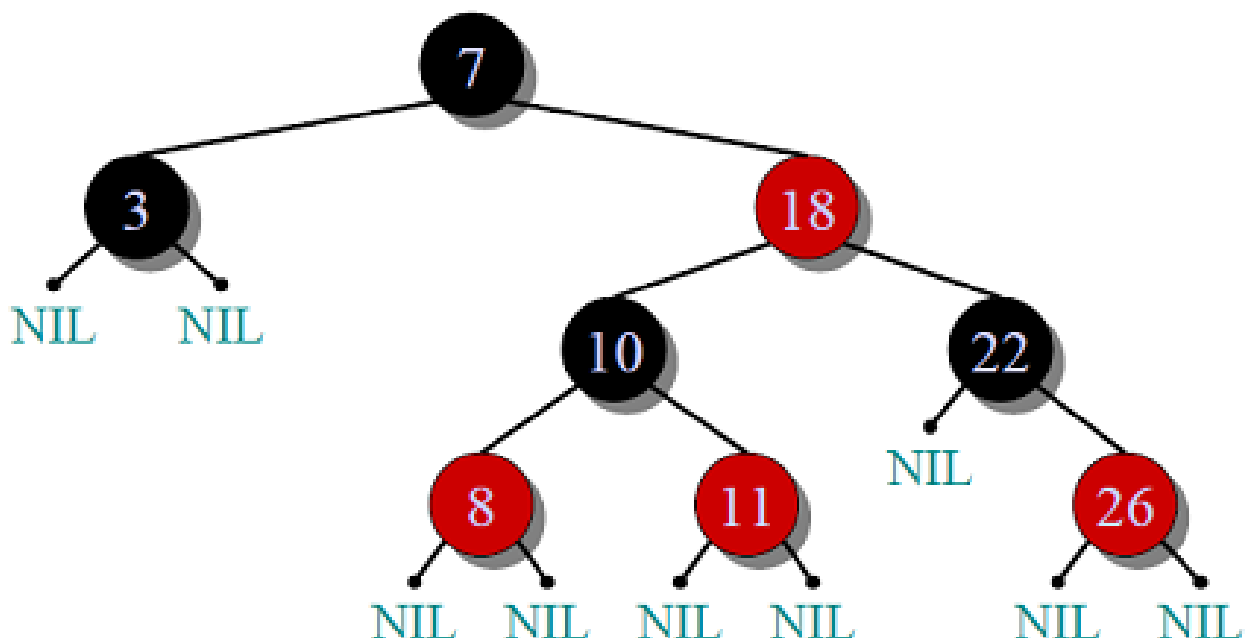


Figure 13: Red-Black Tree

These constraints enforce a critical property of red-black trees: the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly height-balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red-black trees to be efficient in the worst case, unlike ordinary binary search trees.

To see why this is guaranteed, it suffices to consider the effect of properties 4 and 5 together. For a red-black tree T , let B be the number of black nodes in property 5. Let the shortest possible path from the root of T to any leaf consist of B black nodes. Longer possible paths may be constructed by inserting red nodes. However, property 4 makes it impossible to insert more than one consecutive red node. Therefore, ignoring any black NIL leaves, the longest possible path consists of $2*B$ nodes, alternating black and red (this is the worst case). Counting the black NIL leaves, the longest possible path consists of $2*B-1$ nodes.

3.1.6 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

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

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.

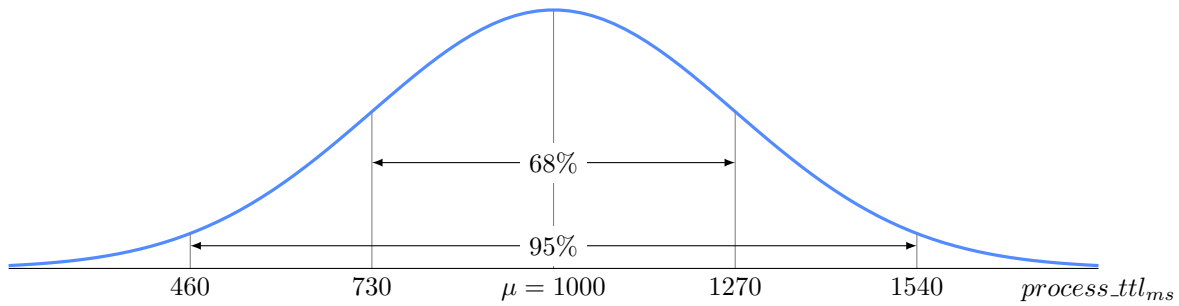
- Process Time to Live.
- Process Number of IO Operations.
- Process IO Operation duration.

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

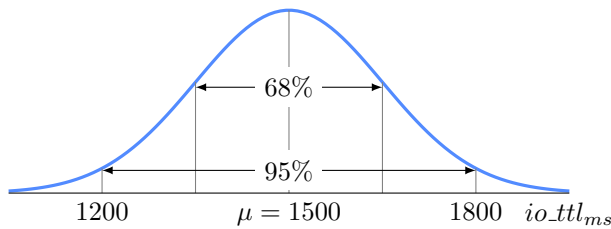
Table 13: Distributions of Process variables

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:

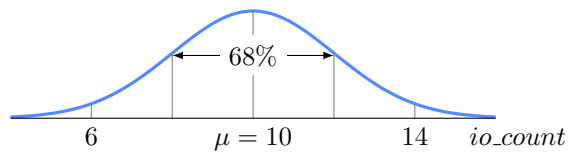


(a) Process TTL Distribution

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, and 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 does nothing more than converting that number into 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 14: 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.

3.1.7 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 ← evaluated()*

```

1: while pool ≠ empty do
2:   screen.update()
3:   if process ← IO then
4:     EXECUTE( $\frac{time\_q}{2}$ )
5:     dispatcher :: interrupt
6:   else
7:     EXECUTE(time_q)
8:     dispatcher :: context_switch
9:   end if
10:  awt ← calculate()
11: end while
12: 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()* → *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

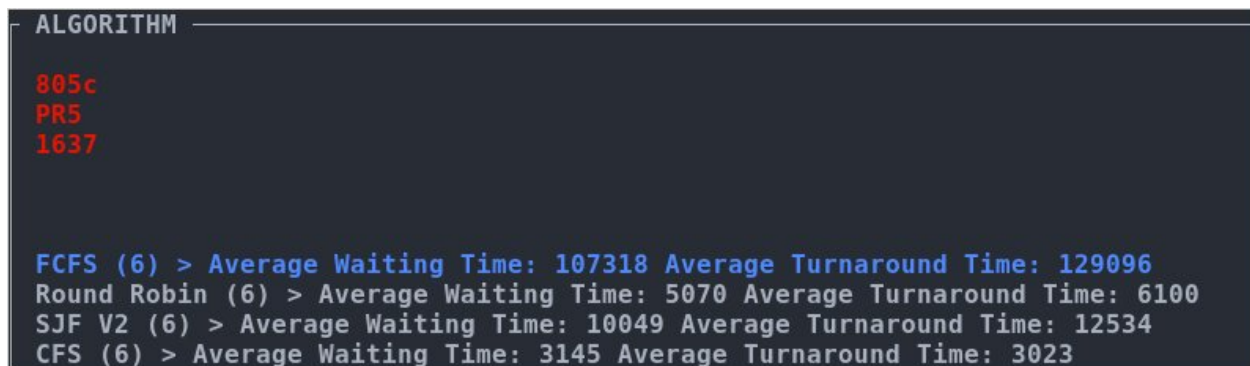
have any process in them. This means that the `waiting_queue` can have a size of 0, while the ready one has all of the processes in it. And vice versa, the `ready_queue` can be completely empty while all of the processes are doing some sort of IO (although this situation is highly unlikely).

3.1.8 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.



```
ALGORITHM
805c
PR5
1637

FCFS (6) > Average Waiting Time: 107318 Average Turnaround Time: 129096
Round Robin (6) > Average Waiting Time: 5070 Average Turnaround Time: 6100
SJF V2 (6) > Average Waiting Time: 10049 Average Turnaround Time: 12534
CFS (6) > Average Waiting Time: 3145 Average Turnaround Time: 3023
```

Figure 16: Algorithm Panel

Here we can see the output of several algorithms with a process running. We can see how the processes is colored into a priority and how the summaries of the previous algorithms are shown to the user. The most recently run algorithm is also colored so it makes it easier to remember which is the most recent piece of data.

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.

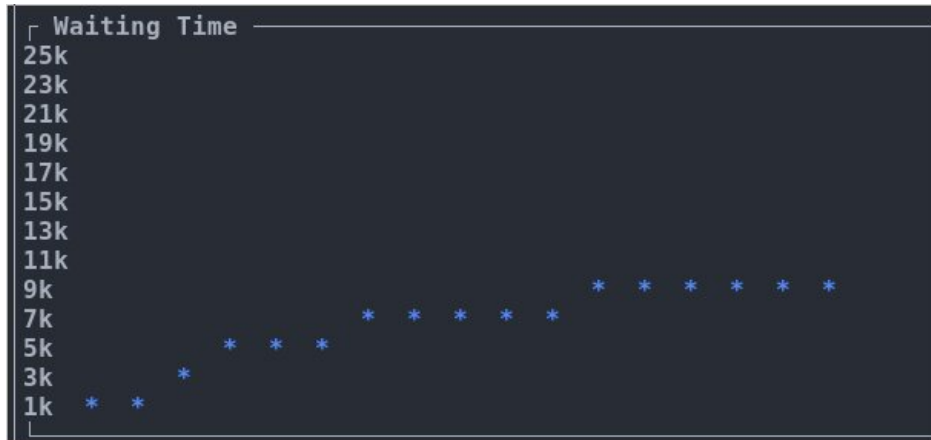


Figure 17: Waiting Time Panel

Here is the screen for the graphical output of the average waiting time. We can see that on the left we have a scale that depicts the running time in milliseconds and it also plots the dots in the window. In this specific case, we can see the graph being very close to a logarithmic function. This is a big help when we are evaluating the algorithms in the end.

- **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 at 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.

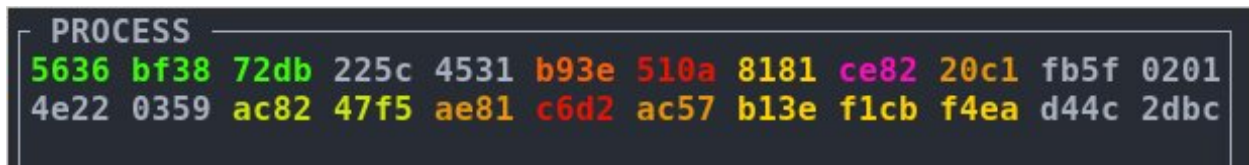


Figure 18: Process Panel

In this screen we can see how the different processes and mix of processes are all in the ready queue. This means that all of these are either about to be executed or are being executed in the moment. We can see that some of the processes are already with a pre-defined priority, while the white ones will get theirs as soon as the algorithm is started.

- **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.

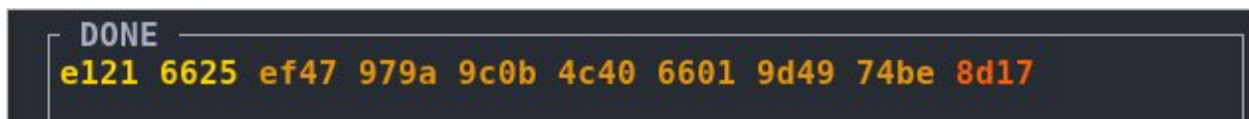


Figure 19: Done Panel

In this panel we can see a set of processes that have already finished execution and are put in the order by which they have finished.

- **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 from 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.

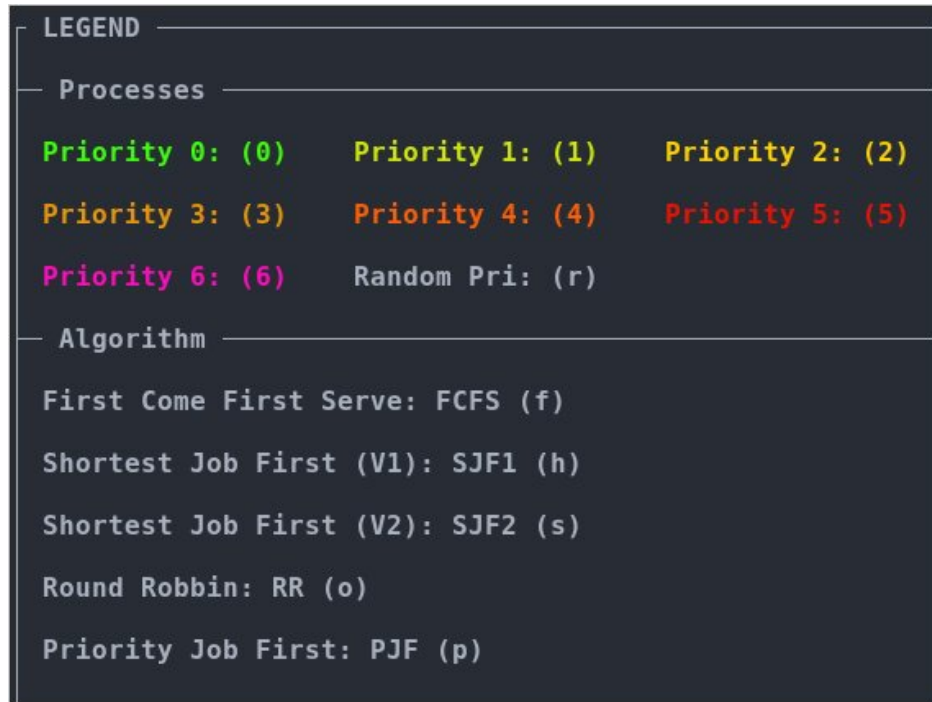


Figure 20: Legend Panel

The legend panel color coded and we can see that in the brackets, next to each procedure, there is a character that will initiate it. Then, we can see how there are two sub-panels, one for the processes and one for the algorithms. This makes it exactly clear to the user what are the functionalities of the application.

On top of everything, there is also additional information that is given for each algorithm that is being currently run. This information can be used to understand what is happening currently on the screen and to also make it easier for any pedagogical lessons.

```
Algorithm

First Come First Serve: FCFS (f)

Shortest Job First (V1): SJF1 (h)

Shortest Job First (V2): SJF2 (s)

Round Robbin: RR (o)

Priority Job First: PJF (p)

Completely Fair Scheduler: CFS (c)

The scheduler assigns a fixed time unit per process,
and cycles through them. If process completes within
that time-slice it gets terminated otherwise it is
rescheduled after giving a chance to other processes.

RR scheduling involves extensive overhead, especially
with a small time unit.

Balanced throughput between FCFS/ FIFO and SJF/SRTF,
shorter jobs are completed faster than in FIFO and
longer processes are completed faster than in SJF.

Good average response time, waiting time is dependent
on number of processes, and not average
process length.

Because of high waiting times, deadlines are rarely
met in a pure RR system.

Starvation can never occur, since no priority is
given. Order of time unit allocation is based upon
process arrival time, similar to FIFO.
```

Figure 21: Legend Info Panel

Here we can see that the text is placed in the free area of the application in the Legend panel and is currently explaining about the Round Robin algorithm. This looks both nice visually and is helpful at the same time.

Because the user interface has to also have some form of input, we must also accommodate to the fact that we must enter some data into the application. This means that we must parse some input from the user to the different processes into the app. This is done through the ncurses library, which allows for inputs. With the `getch()` method, which returns a number representation of the pressed character from the keyboard, we can see what the user has pressed. By following the legend and having the input given by the user, we can see what is needed to be done in the app. For instance, we might start executing some algorithm or we might create a specific process.

The legend for the different accepted types of input is as follows:

- 0 to 6 . Pressing one of the numbers from 0 to 6 will create a certain type of a process, which is based on a priority. The priority on each of the button presses is expressed by a graphical color, indicated in the legend. When a process is created this way, the process with its ID is added to the ready queue and is printed with the priority color immediately.

- **r** . When this key is pressed, then a new type of process is created - a random one. Based on the aforementioned rules for process distribution, a new process is added to the ready queue and is colored in white, because then the algorithm will evaluate the process priority. It is completely possible for a user to both combine and mix random and pre-defined processes in the ready queue, which would cause the queue to be filled with colored and non-colored processes.
- **f** . When pressing the F key, we start working with the different algorithms. In this case, the FCFS algorithm would start executing on the filled ready queue. This means that we will start seeing how the processes we just created will be treated by this algorithm.
- **h** . As with the previous key, this will start the SJF method 1 algorithm.
- **s** . This is the SJF method 2 implementation.
- **o** . This is the Round Robin algorithm key.
- **p** . This is the Priority First scheduler.
- **c** . This is the Completely Fair Scheduler in action.

3.1.9 Software Architecture

The Software Architecture of the project can be seen as one of the following: *MVC*, *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` → `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.

MVC

The Model View Controller (usually known as MVC) is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

The Model View Controller approach in this project can be seen as the following way. First, we can check what is the Model of the application. This is the part where the data is stored and is handled by the business logic. The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application. The model is responsible for managing the data of the application. It receives user input from the controller. A large part of the app is in the Model section, because it is handling and working with a lot of algorithmic data. So we can consider that the Dispatcher, Scheduler, the Red-Black tree, the Process Poll, Processes, etc. Because all of this data is working in the background and we just see its visualization, we can put it in the Model. All of the algorithms that are implemented, plus the management of the data-structures and the pool queues are also handled in the Model part.

The View is for any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The view means presentation of the model in a particular format. For us, we have plenty of data in the View, although we have

only one class that is managing this whole section, which would be the screen class. This class manages all of the windows and sub-windows that we see, plus any text and dynamics that are represented during execution. This is great for us, because we have one singleton that has an open API to the other parts, and can be called and used in a completely decoupled way. So if we want to change some of the headings on some of the panels or to create some new data, the screen class allows us to do that easily. Beneath the screen class lies the ncurses library that allows all of these graphics manipulations in the terminal, but having a custom written API makes things much more easier to manage and work with. In the end we end up with an intuitive and fun User Interface that makes working with the application much more enjoyable.

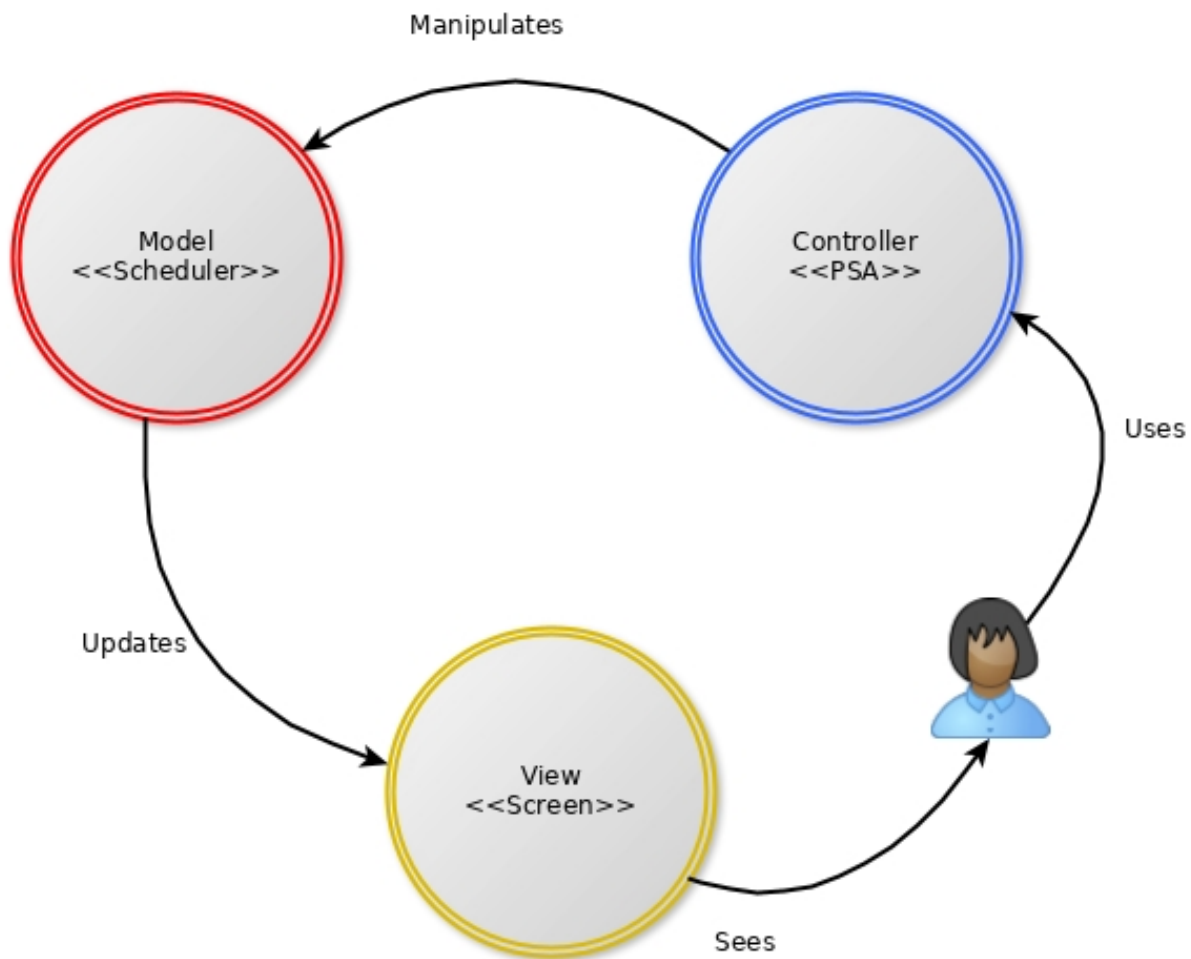


Figure 22: MVC Diagram

From this MVC diagram we can see that the user is interacting with the Controller part, which is the driver program of the PSA class. Then that class manipulates the Model, which we can sum up with the Scheduler class. Finally, the Model is constantly updating the View part, being that the Screen class. The Screen class gives the visual information to the user and then the circle is complete.

The Controller accepts input and converts it to commands for the model or view. The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model. For this application, the Controller is the main driver of the

project, which would be just a small part, actually the smallest piece of code, that there is, that manages the data flow. In this case, the controller is the one that takes in the data character input from the user and sends it to the scheduling system, which would be part of the Model. Then the Controller updates the information and gives it to the screen, where the View can add it to the output and we can see it represented in the terminal. Although very small, it is a key part of the whole system, because it accounts for security measures and proper data flow.

Monolithic

A Monolithic application describes 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.

Of course another approach to take the architecture of the project is to look at it like a monolithic structure. Because of the overwhelming part of the logic and algorithmic work that is being put into this app, we can say that it is massive in those terms and that it holds value as a desktop application. The GUI part of the desktop app is the terminal and the terminal graphics that are being used. Then all of the other data is being processes is part of the monolithic structure and comes packaged together into a whole bundle. This doesn't mean that the project is unmanageable, or that it is difficult to write and support, but it means that it has built up to a stature of complexity, similar to that of a desktop app.

Because the app is written in C++, a language which, although popular, is difficult to distribute across many platforms, and is also using a third-party software library (the ncurses library), we can say that packaging this app into a single entity makes sense and provides good distribution and sharing across platforms. With a simple installation, this can be ported to other Linux boxes, where it can be freely run. This also means that it can be used in a classroom, where the students can open the application and play with it for learning purposes.

3.1.10 Components

From a component standpoint, we can see in the following diagram how the whole procedure in the application fairs. There are two interesting things to notice in the diagram.

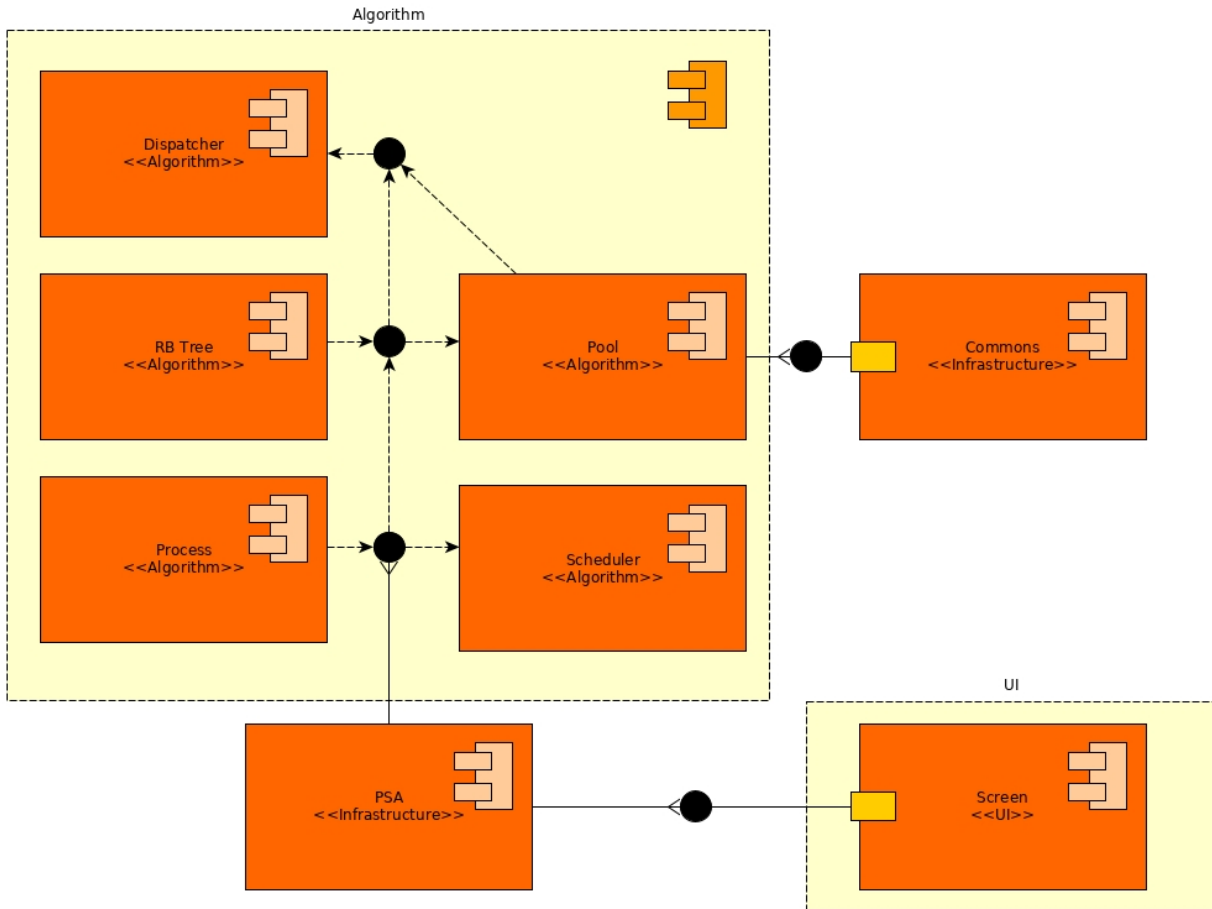


Figure 23: Component Diagram

First thing is that the diagram is relatively simple to follow. We have 3 major components that are separated into their respective place. There is the Algorithm part, which encapsulates most of the classes in the app. There is the UI component section, which comprises from the screen class. Finally, there is Infrastructure section, which can be seen in the two separate classes - the PSA and the Commons section.

The second thing we cannot directly see from this diagram is that although it looks simple, the internal workings of the application are hidden from the diagram. The meat of the project is hidden behind the algorithm, trees, dispatchers, pools, and processes, which all have a high dynamic of data flow.

Let's look at each component individually. Starting from the Screen, it is the part of the application which ensures the UI is working. This is the only a single component, because we don't need it to be too big or too complex. It has an open interface towards the driver section of the app. The screen is the class that takes every piece of data and provides a nice output for the terminal. Thus, this is the way we see things that are put on the screen. Now let's move to the Infrastructure part. This comprises of two separate classes. The reason they are not under the same umbrella, is because they play a different key role in the app, but are still a vital part of the whole functionality. The Commons class is a generic small library, that includes different pieces of data and calculations that do not fit in any other place. For instance, it calculates the different distributions for the process generations, it provides conversions from numbers to strings, and so on. This component also has an open API, which allows very easy access to the different functionalities. The PSA is another small component, which is the main driver

of the app. Through it, we reach the UI section in order to regularly update the screen, input key presses, and also start new algorithms and add new processes. The final part of the whole story is the Algorithm section. This is by far the largest piece of software we have. First we take a look at the Pool. This has a direct access to the Commons component, because it uses it the most on a regular basis. Then the data flow from the pool is going through the Dispatcher, then onto the Scheduler and Process classes. All of these classes are simple to understand, but the magic hides behind the fact that they are calculating complex things. We also have a Red Black tree section, which exists only for the CFS algorithm, but the data flow also passes through it.

So in conclusion for this part, we can say that the app is easy to understand, but much more difficult to follow once the algorithms start executing.

3.1.11 Deployment

From a deployment stand point, we have an even simpler diagram, but that's okay. This is still a desktop application and a project that serves to be pedagogical and used for research. So let's look at the diagram.

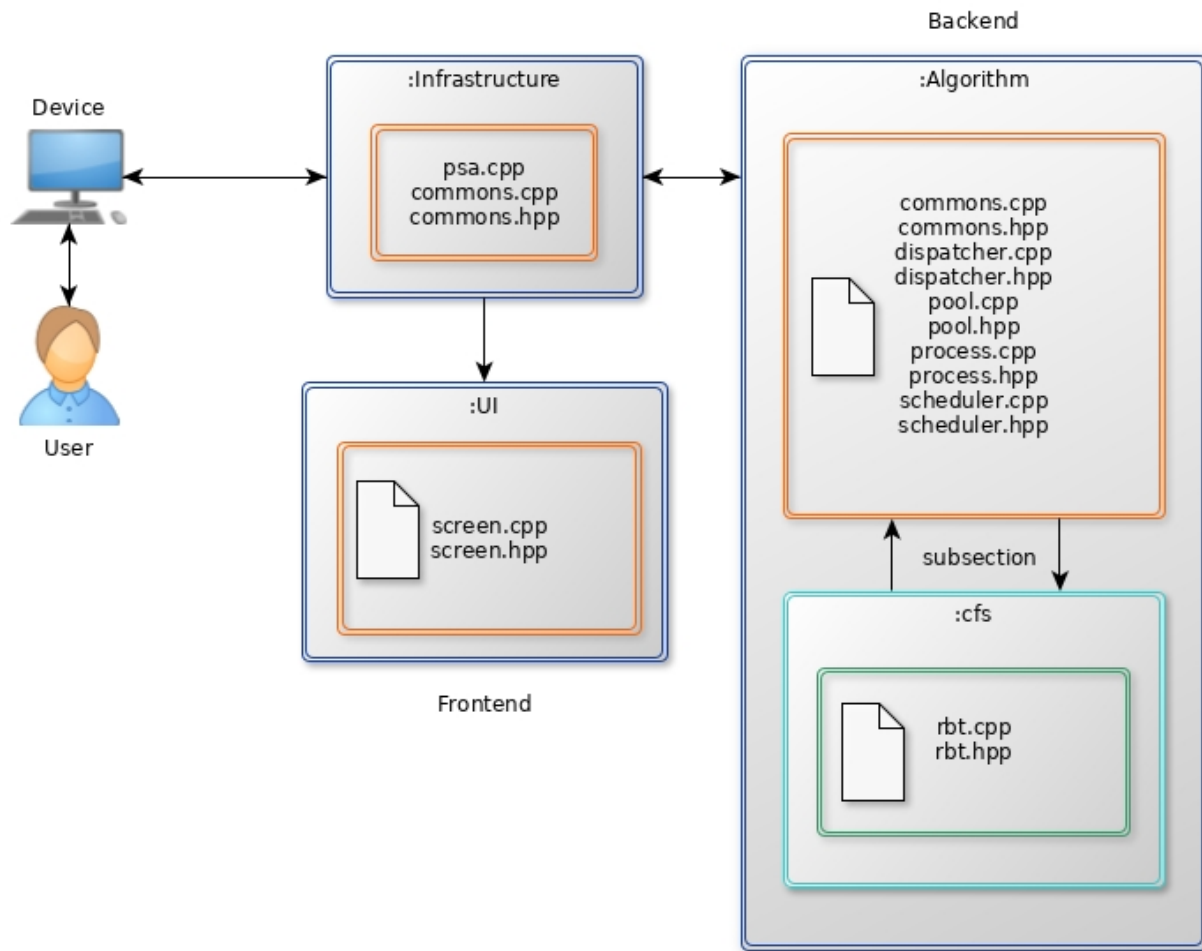


Figure 24: Deployment Diagram

Here we can see how the user can access the project through some machine and can run it there. Once we start,

we go to the infrastructure part, where we can see which files play a role in. These would be the psa and common files, allowing for a connection between the other components. In this case we can see that this is the Controller section of the app looking at it like an MVC. Then there is the Frontend part, which is the screen files, allowing for ncurses to do its job. The Backend is part of an algorithm section, or the Model if you like. Then we have two subsections - one is the general bunch of algorithms that are part of the whole project and another is the subsection for the CFS scheduler, where we can see that the red black tree files are there. So these two comprise the back end of the project and it communicates with the psa file in order to provide everything to the user.

Evaluation

Pros	Cons
Used in real systems	Very hard to implement
Super Fast	
Optimal	
Interesting	

Algorithm Complexity	Execution Complexity
$O(n)$	$O(\log(n))$

Table 15: Pros and Cons of PJF

This algorithm is very interesting and great to test and play with, but it is very difficult to understand and even implement, because of the difficult data structures, variable management, tracking and even visualization. This is my favorite algorithm from the whole bunch, but it took so long to implement that a lot of stress was built up around it. It was worth it regardless!

3.1.12 Security Features

Coming into the security part of the application, there are a few things to consider. First, because this is a desktop app and it does not have an actual database or a connection to the internet, we can say that the security measures we have to take must be unconventional. Second, we must be careful with the execution of the processes, as this is a desktop app, we must make sure that we do not stall the terminal or break the system if we overload it.

This project has a few files that are loaded and used to represent the information for the different algorithms. These files come with the standard package of the program and they are part of the source files, where they are programmatically loaded. So unless someone manually start tempering with the project system files, in order to change to algorithm output when it is given in the legend panel, then there is no need nor necessity for any more complex database preservation or encryption schemes. Additionally, the program is written in such a way, that the process execution and creation cannot be tempered with through any breakage and cannot be changed in any configuration files. For one, there are not configuration files to begin with. The best security is the one for which you don't need any security.

Another aspect we have to look at is the distribution method for the project. Because we want this to be used for research and studying, we have to have it in a safe space and packaged without being compromised. This can easily be achieved if there is a distributed source and package that is downloadable over a secure connection. One such way to achieve this is to upload the project to GitHub and to package it systematically for different

operating systems. This both allows to have an open source secure project that can be assured that there is no tempering with and it also allows for easy usage of the app in general.

We must also keep in mind that because we are playing with timing, processes, simulations, and also having the risk of blocking the whole terminal, we must take special precautions that do not allow any such situations. One thing to notice is that there are some cases where if we do not limit the user interface, then the user can destroy their system. So what could happen is that they can overload the application, which could cause the terminal to overload and then the CPU start overheating. Because of this, the user is allow to have a limited input window, there are no open inputs that would cause overheating and needless input, there is not a chance to break the system and it's safe for the common user.

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`. There isn't much of a importance on what Linux box this is tested and what terminal is used, as long as the app has access to the `ncurses` library that is used.

The reason that `C++` was used is because it's both a challenging and granular language. `C++` is very verbose and you have to write a lot of code to get things going, there is a lot of file management, but it also allows for much faster and better performance. If we compare this to using Python, then Python is fast, but not as fast as `C++` and it doesn't allow for such fine tuning that `C++` has. Because of the nature of the project, where we are moving objects, working with move semantics, have some form of memory management, dealing with mutexes, and having additional threads, then `C++` is the right choice.

Installation

In order to run this project, we must have the `ncurses` library, we must have a running terminal emulator, such as `gnome-terminal` or `xterm`, and we must have `C++ 11` or above running. Then we acquire a copy of the project, optimally from GitHub, and once we download it, we just have to compile it. The compilation step is very easy, because in order to do that we just have to run one single command. The reason for this is because there is a nifty `Makefile`, where all of the different files are bundled and combined, which makes a nice executable in the end.

```
1 # Compiler:
2 CC = clang++
3
4 # Compiler flags:
```

```

5 # -g          adds debugging information to the executable file
6 # -std=c++11   use c++ 11 standard
7 # -Wall        turns on most, but not all, compiler warnings
8 # -lncurses    adds the ncurses library
9 CPPFLAGS = -g -std=c++11 -Wall -lncurses -lpthread
10
11 # Invoke the first target entry:
12 default: psa
13
14 # To create the executable file:
15 psa: psa.o process.o commons.o screen.o scheduler.o pool.o dispatcher.o rbtree.o
16     $(CC) $(CPPFLAGS) -o psa.app psa.o process.o commons.o screen.o scheduler.o pool.o
17     dispatcher.o rbtree.o
18
19 # Provide source files:
20 psa.o: psa.cpp
21     $(CC) -c psa.cpp
22
23 process.o: process.cpp
24     $(CC) -c process.cpp
25
26 commons.o: commons.cpp
27     $(CC) -c commons.cpp
28
29 screen.o: screen.cpp
30     $(CC) -c screen.cpp
31
32 scheduler.o: scheduler.cpp
33     $(CC) -c scheduler.cpp
34
35 pool.o: pool.cpp
36     $(CC) -c pool.cpp
37
38 dispatcher.o: dispatcher.cpp
39     $(CC) -c dispatcher.cpp
40
41 rbtree.o: cfs/rbtree.cpp
42     $(CC) -c cfs/rbtree.cpp
43
44 # Remove executable files, as well as old .o object
45 # files and *~ backup files:
46 clean:
47     $(RM) psa *.o *~

```

With this file, we can run a command and we can see that the product would be an executable that is called `psa.app`. It is clear from the file that we are using at least C++ 11, with the pthread library, the ncurses library, with all the warning are on.

1 # Download source

```

2 git clone https://github.com/Anarcroth/process-scheduling-analysis.git
3
4 # Navigate to folder
5 cd process-scheduling-analysis
6 cd src
7
8 # Compile
9 make
10
11 # Run
12 ./psa.app

```

All of these steps have to be done in the terminal. Then when we execute them, we will be sure that we will have a running application. It will be in full screen and we can start using it.

Interesting Implementations

There are a few implementations that are curious and worth looking at. First off, we can look at the algorithm implementations for different algorithms. There is no point in looking at every algorithm, but there are a few that are interesting. So we will look at the SJF method 2, Round-Robin, CFS.

So once we know how these algorithms work, we can see their implementation in action. Looking at the implementation of SJF, we have to have some form of predictions and we have to also track which would be the shortest processes.

```

1 void scheduler::sjf_v2()
2 {
3     PSAscreen::get().show_alg_info("SJF");
4
5     reset();
6     pool::eval_prcty();
7
8     int prediction = prev_pr_burst;
9     auto pit = pool::ready_queue.begin();
10    while (pit != pool::ready_queue.end()) {
11        PSAscreen::get().update_process_scr(*pit);
12
13        // must be saved before the next process comes
14        int prev_pr_ttl = pit->get_ttl();
15        take(pit, prediction);
16        prediction = exponential_average(prev_pr_ttl);
17    }
18    add_summary("SJF_V2");
19    PSAscreen::get().show_statistics(summaries);
20 }

```

In the beginning, we show the info for the algorithm. Then we reset the values for which we keep track of, such as the global statistics. Each process has its priority evaluated. Then an execution estimation is made as to how long should that process execute. This prediction is updated on every that comes next. Each process can have IO and be pushed to the ready queue again. At the end, the statistics of executions are shown.


```

1 int scheduler::exponential_average(int prev_pr_ttl)
2 {
3     prev_pr_burst = ALPHA * prev_pr_ttl + (1 - ALPHA) * prev_pr_burst;
4     return prev_pr_burst;
5 }

```

This calculates the next expected duration of a process CPU burst and returns it as a recommendation. As we know from the aforementioned theory above, then we know that the constant `ALPHA` must be somewhere between 0 and 1, with a middle value of 0.5. In this case, we use the middle value because we both care for the previous process and the processes before that. Of course this constant is made as a part of the scheduler, which can be programatically changed.

```

1 void scheduler::round_rob()
2 {
3     PSAScreen::get().show_alg_info("RR");
4
5     reset();
6     pool::eval_prcls_prty();
7     auto pit = pool::ready_queue.begin();
8     while (!pool::empty()) {
9         PSAScreen::get().show_process(*pit);
10        take(pit, TIME_QUANTUM);
11    }
12    add_summary("Round_Robin");
13    PSAScreen::get().show_statistics(summaries);
14 }

```

Each process has its priority evaluated. Each sequential process is taken and is executed for a constant `TIME_QUANTUM`. Each process can have IO and be pushed to the ready queue again. At the end, the statistics of executions are shown. As with the previous algorithm, we have another global constant that is put in the scheduler class. This `TIME_QUANTUM` is set to the middle value of having 50 milliseconds. For us, this is just fine, because with this time, we are forcing the program to sleep for that amount of time. Although we can make it sleep for less time, this sleeping function causes quite a bit of overhead, which means that if the number is slow enough then we would be causes an un-even process execution, stutter, and jitters in the graphics.

```

1 void scheduler::cfs()
2 {
3     PSAScreen::get().show_alg_info("CFS");
4
5     reset();
6     pool::eval_prcls_prty();
7
8     rbtree rbt;

```

```

9   for (auto p : pool::ready_queue)
10      rbt.insert(p);
11
12   // small hack to make the thing more optimal
13   rbt.fix_internal_repr();

```

Finally we are looking at the CFS algorithm. It is a hefty piece of software, but what we can take away is that we have one additional object, which would be the red black tree. We have to do a few hacks in order to make things a little bit better. Because until now we were dealing with the global pool which was holding the processes, introducing a whole other complex tree structure to the mix causes huge problems with visualizing the algorithm. This means that we need an internal set of vectors that will be holding the processes in the tree. That is why we must maintain two queues in order for things to work. Because of this additional maintenance we have to do, we must optimize things a little bit, which means that we have to keep a proper organization of the processes in the tree queues. That is why we are sorting the processes once we enter them in the tree initially.

```

1   while (!rbt.empty()) {
2
3       sched_entity *shortest = rbt.get_smallest(rbt.root);
4       process pr = shortest->key;
5
6       rbt.delete_node(shortest);
7
8       PSAScreen::get().update_process_scr(pr);
9
10      // stats
11      pr.set_tos(total_t);
12      pr.add_wait_t(total_t);
13
14      // exec time calc
15      pr.calc_max_exec_t();
16      int exec_t = 0;
17      if (pr.get_max_exec_t() > 0)
18          exec_t = pr.get_max_exec_t();
19      else
20          exec_t = pr.get_ttl();
21
22      // IO
23      if (pr.has_io())
24          exec(exec_t / 2);
25      else
26          exec(exec_t);
27
28      pr.add_vruntime(exec_t);
29      dispatcher::cfs::con_swch(pr, exec_t, rbt);
30  }
31 }

```

And while we are iterating over the non-empty tree, we have to pick the left-most node of the tree. This will be the smallest node in the tree, which is the exact one we want to get. After we get the process, we delete that

node, as we have to do in theory, and then we continue with the execution of the algorithm. We do the standard procedures of accumulating the statistical data for each process and we also take into account the fact that must execute the maximum execution time. Due to the limitations of this project, we make some calculations a little bit simpler so that we do not waste time and still have a working version. After we calculate the maximum execution time, we then push the process to be executed for that amount and then we add it to the virtual runtime. In the end we call the special dispatcher, which does a context switch.

Now that we have mentioned the dispatcher, we can look at how it works and what are the curiosities around it. We know that the two main jobs of this piece is to either engage into context switches or to handle interrupts. Here is how the interrupt handler works.

```
1 void interrupt(std::vector<process>::iterator& pit, int tq)
2 {
3     save_state(pit, tq);
4
5     pool::wait_queue.push_back(std::move(*pit));
6     pit = pool::ready_queue.erase(pit);
7
8     std::thread iothread(dispatcher::exec_io,
9                           pool::wait_queue.begin());
10    iothread.detach();
11 }
```

We can see that even during such an event, we first must save the state of the current process, expressed as a pointer iterator to the actual process. Then we push that process to the waiting queue. Here we are utilizing the move semantics of C++, which basically transfers ownership of one object from one container to another. We remove the process from the ready queue and then we create a special separate thread which will call the execution of the IO operation. Then what we do is we detach this separate thread from the main one. By doing this, we tell C++ that we want this method to execute independently from any other execution and when it finishes, it will end quietly. In order to do this we need the following code.

```
1 void exec_io(std::vector<process>::iterator pit)
2 {
3     iomutex.lock();
4
5     auto io_ttl = pit->get_ioops().begin();
6     std::this_thread::sleep_for(std::chrono::milliseconds(*io_ttl));
7     pit->add_tat(*io_ttl);
8     pit->get_ioops().erase(io_ttl);
9
10    pool::ready_queue.push_back(std::move(*pit));
11    pool::wait_queue.erase(pit);
12
13    iomutex.unlock();
14 }
```

Here, using the special `iomutex` object, we lock this piece of code and ensure that these calculations that we are doing are not going to cause a race condition and that we won't invalidate some piece of memory. Additionally, here we are executing the IO operation, by calling the sleep function and removing one IO operation from the set that the specific process has.

Another interesting thing that we have to look at is the famous red black tree, which was a great achievement to program in C++. Here we can see in the snippet that there are some curiosities with the insertion and deletion sections.

```
1 void rbtree::insert(sched_entity *&node, sched_entity *&parent, process key)
2 {
3     if (!node) {
4         node = new sched_entity(key, parent, nullptr, nullptr, col::RED);
5         rebalance(node);
6         rq.push_back(key);
7     } else if (key.get_vruntime() <= node->key.get_vruntime()) {
8         insert(node->left, node, key);
9     } else {
10        insert(node->right, node, key);
11    }
12 }
```

This is the main insertion function that creates new nodes, based on the calculated virtual runtime of each process. When a process is deleted and then re-inserted with the new runtime, this method does all the magic. Here we can see the recursive nature of the method, where the empty left and right nodes are being created when we reach the right position with the key. We also see two interesting methods. One is the push back to the internal ready queue, which is necessary to do here, because as we already mentioned, with this structure we need to maintain such queues separately. The other thing we notice is the rebalance method, which is the next code we cover.

```
1 void rbtree::rebalance(sched_entity *&node)
2 {
3     if (!node)
4         return;
5
6     if (root == node) {
7         root->rb = col::BLACK;
8         return;
9     }
10
11    sched_entity *nnode = node;
12    if (node->rb == col::RED && node->parent->rb == col::RED) {
13        auto *grand_p = parent(node->parent);
14        sched_entity *aunt;
15
16        if (grand_p->left != node->parent)
17            aunt = grand_p->left;
18        else
```

```

19     aunt = grand_p->right;
20
21     if (!aunt || aunt->rb == col::BLACK)
22         nnode = rotate(node, grand_p);
23     else
24         nnode = color_flip(node);
25 }
26
27 // if there is no parent to the root
28 if (!nnode->parent)
29     root = nnode;
30
31 rebalance(nnode->parent);
32 }

```

The rebalance method is very important to have, because of the complex rules that the insertion has. Here we do a lot of things. First we check that the node we are inserting is valid. If it is not, we quit. Then if that node is equal to the root node, we have to make sure that the root is black and then we don't need to do anything else so we return nothing. After that, we check if the node and the parent node are red, which means that we have two consecutive nodes one after another. If we have such a case, then we take the grand parent of the node, then we check the state of the aunt. If the aunt is black or null, then we have to do a rotation in order to rebalance the tree properly. Otherwise, we just flip the colors. In the end we make sure that the root is the right one. Finally, we ensure the rest of the tree is valid and call the same method recursively.

```

1 sched_entity *rbtree::left_rot(sched_entity *&root)
2 {
3     auto *tmp = root->right;
4     auto *grand_p = parent(root);
5     root->right = tmp->left;
6     tmp->left = root;
7     root->parent = tmp;
8     if (root->right)
9         root->right->parent = root;
10    if (grand_p) {
11        if (root == grand_p->left)
12            grand_p->left = tmp;
13        else if (root == grand_p->right)
14            grand_p->right = tmp;
15    }
16    tmp->parent = grand_p;
17    return tmp;
18 }

```

Because rotation is a key point of the whole algorithm, we have to see how such a rotation is done. It is very similar to the rotation done in an AVL tree, but there are a few other complications. Because the red black tree is having the notion of parent and aunt nodes, we must keep a valid pointer track for the parent nodes of each node. Because of this thing, we have to check a few cases, where the parent node after rotation should be fixed and in the proper place. In this method, we can see that the standard rotation is done in the first 4 lines, but after that we have to check if the grand parent is valid and then assign the proper children to it.

```

1 void rbtree::__delete_node(sched_entity *node)
2 {
3     sched_entity *rnode = replace(node);
4
5     bool dubblack = ((!rnode || rnode->rb == col::BLACK)
6                     && (node->rb == col::BLACK));
7
8     sched_entity *par = parent(node);
9
10    if (!rnode) {
11        if (node == root) {
12            root = nullptr;
13        } else {
14            if (dubblack) {
15                fix_dubs_black(node);
16            } else {
17                if (sibling(node))
18                    sibling(node)->rb = col::RED;
19            }
20
21            if (node == node->parent->left)
22                par->left = nullptr;
23            else
24                par->right = nullptr;
25        }
26        delete node;
27        return;
28    }

```

The deletion part of the tree is also quite difficult to get, because the red black tree has a notion of being in a state of a double blacks, which has to be fixed in order for the balancing to work. There is no point in showing all of the code, but we have to look at least some of the implementation, to get an idea of what's happening. So in the beginning, what we do is we check for the double black case, then we replace the node with the node we passed as an argument, and then we proceed with the balancing.

```

1 int process::set_ttl()
2 {
3     // Guarantees no negative ttls
4     int temp_ttl = 0;
5     while (temp_ttl < 1)
6         temp_ttl = commons::gen_gaus_rand(TTL_MEAN, TTL_STDDEV);
7
8     return temp_ttl;
9 }

```

An interesting thing to also consider is how the process time to lives are created. We use the commons library in order to generate the normal distribution with which set the ttl of the specific process. The two constants we

see are part of the process global constants and are there so that every process is with the same relative numbers in the end. Also this method guarantees that there won't be any outliers or at least any that are with negative numbers, which would be complete nonsense.

5 Testing

Testing this application is done through several executables that have been created in order to generate several procedural programs that would in the end evaluate the average performance of each algorithm. These exes have been made in order to streamline and make evaluation of the algorithms easier in the end. In addition to testing the different algorithms, also testing the effects of the different types of input a user has on the app was also done.

A part of the initial executables can be portrayed like so.

```
1 -rwxr-xr-x 1 root root 228K 03:03 psa100fcfs.app
2 -rwxr-xr-x 1 root root 228K 03:02 psa100rr.app
3 -rwxr-xr-x 1 root root 228K 03:02 psa100sjf1.app
4 -rwxr-xr-x 1 root root 228K 03:03 psa100sjf2.app
5 -rwxr-xr-x 1 root root 228K 03:04 psa200fcfs.app
6 -rwxr-xr-x 1 root root 228K 03:02 psa200rr.app
7 -rwxr-xr-x 1 root root 228K 03:02 psa200sjf1.app
8 -rwxr-xr-x 1 root root 228K 03:03 psa200sjf2.app
9 -rwxr-xr-x 1 root root 228K 03:01 psa50fcfs.app
10 -rwxr-xr-x 1 root root 228K 03:01 psa50rr.app
11 -rwxr-xr-x 1 root root 228K 03:01 psa50sjf1.app
12 -rwxr-xr-x 1 root root 228K 03:01 psa50sjf2.app
```

This is the output if we list part of the generated test apps. They represent a few things. The number indicates the number of processes used in the experiment. Then after the number, follows the name of the algorithm that is being tested. In the end, and output is generated to the standard output to the terminal and it looks like this.

```
1 Round Robin (50) > Average Waiting Time: 42645 Average Turnaround Time: 42450
2 Round Robin (50) > Average Waiting Time: 41640 Average Turnaround Time: 41460
3 Round Robin (50) > Average Waiting Time: 42418 Average Turnaround Time: 42290
4 Round Robin (50) > Average Waiting Time: 43167 Average Turnaround Time: 42999
5 Round Robin (50) > Average Waiting Time: 42914 Average Turnaround Time: 42734
6 Round Robin (50) > Average Waiting Time: 43048 Average Turnaround Time: 42864
7 Round Robin (50) > Average Waiting Time: 40998 Average Turnaround Time: 40795
8 Round Robin (50) > Average Waiting Time: 41722 Average Turnaround Time: 41525
9 Round Robin (50) > Average Waiting Time: 44473 Average Turnaround Time: 44334
10 Round Robin (50) > Average Waiting Time: 42477 Average Turnaround Time: 42280
11 Average Wait Time: 42645
12 Average Turnaround Time: 42450
13 Average Time of Execution: 52546
```

We have to keep in mind that this output is the generation of running the specified algorithm with the number of process in total of 10 times, where the total output is averaged between all 10 runs. The reason this was done

is because we are generating random numbers, which means that we want to avoid having one set, which might have an uneven process distribution for some reason in comparison to another set.

Another set of important tests that were done were in order to make sure that the app will not break under normal usage. During the course of development, there were cases where the process would continue indefinitely and will stall the whole terminal window. This was a big problem, as the terminal had to be forced to be stopped and closed. This caused a lot of problems, so in order to be fixed, not only the expected types of input were tested, but also all types of inputs, key presses and combinations and so on had to be checked in order for things to be running smoothly.

Because we are writing in C++, we always fall into the trap that we would be having invalid memory allocations and that there are some cases that the program will break. Because of this, test cases were made for the red black tree which were specific for it and that it functioned normally. This means that two things have to be done. Log files were created in the name of passing the test cases and also additional helper functions were generated that also ensured normal work. The log files were procedurally appended with data that was evaluated and then check several times to guarantee proper working mechanics. The two log files were called `log_file.txt` and `log_rbt.txt`. Because we need two different logs for the red black tree and the scheduler, we have these two.

```
1 void log(const std::string &text)
2 {
3     std::ofstream log_file(
4         "log_file.txt", std::ios_base::out | std::ios_base::app);
5     log_file << text << std::endl;
6 }
```

With this simple helper function we can log anything we want and then we can see what is the result. The reason we want log files is two-fold. For one, when C++ spits out a `segfault`, we are not aware what and where the problem is. Also we are working with a graphical interface, which makes things even more difficult to debug. Having these log files was a great idea in order to do proper testing in the end.

Here is one cases that took a lot of time to debug and fix, which was about 5 days of non-stop testing.

```
1 sched_entity *rbtree::left_rot(sched_entity *&root)
2 {
3     auto *tmp = root->right;
4     auto *grand_p = parent(root);
5     root->right = tmp->left;
```

In these simple lines of code, part of the rotation scheme, we have a devilish bug that is fixed with the `parent(node)` method. If that method is not there and we are directly accessing the node parent pointer, then there is some form of compiler optimization happening in the code, which is not validating the pointer and is removing the reference to the parent. This was a very difficult bug to find and fix, but because of logging and acceptance test, I did it successfully!

```
1 sched_entity *rbtree::parent(sched_entity *root)
```



```

2 {
3     return *&root->parent;
4 }

```

With this function, we have a copy of the pointer we want and we also stop the compiler from doing any unnecessary optimizations that were happening under the hood.

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. For instance, we would expect that the FCFS algorithm would fair in a similar fashion to the SJF one, with the first method implemented. Also we would expect that the Round Robin would be a bit faster than the rest. In the end, we would also expect that CFS would be the fastest of them all, meaning that it is the best algorithm to use for the most general cases. This does also mean that we are looking at the execution time and the criteria for waiting time and turnaround time. If the waiting time is low, then the algorithm is doing good, if it is high, then it is not an optimal procedure.

We must also note on what algorithms we are comparing. Some are not so suitable to be compared to others, which means that we must pick the right algorithms to work with, when doing the final evaluation. Having this in mind, let's take a look at the FCFS and the SJF method 1 algorithms to see how they are doing.

FCFS			
<i>Num Processes</i>	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			
<i>Num Processes</i>	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

From this table and the set of processes we can see a few things. One is that for all of the different executions on the different number of processes, we can see that there is a pattern followed by the algorithms. We notice that there is a close to an linear progression of the FCFS algorithm and the SJF. The waiting time increases in the same manner as the process count increases. The turnaround time is relatively the same and there is a good reason for that. As we saw from the Gantt charts, the turnaround time would be close to that of the process time to live. That is why in these cases, it is almost the same, because the process execution time is around the time of 1000 milliseconds.

Our expectations and guesses are correct for these algorithms and are in line with the theory.

The next set of processes we can compare is the SJF method 2 and the Round Robin one. Here we can see that several other factors play in hand. Here we introduce the notion of the processes having IO operations. Knowing this, we can expect that the execution time would be larger and that the numbers would also be. We would also expect that the SJF method 2 algorithm would be a bit better than the previous ones, but still not as good as the Round Robin one. Let's look at the data.

SJF2			
<i>Num Processes</i>	Exec Time_{ms}	Wait Time_{ms}	Turnaround Time_{ms}
20	73043	38126	34787
50	188321	98706	78400
100	368691	202500	159449

RR			
<i>Num Processes</i>	Exec Time_{ms}	Wait Time_{ms}	Turnaround Time_{ms}
20	20593	17512	18125
50	50411	45391	45254
100	192607	81059	79558

Here we can see that there are huge jumps between the two tables. We notice that there are for the SJF 2 algorithm, the execution time and the waiting time are increasing not exactly in linear fashion, but also not in logarithmic time. This means that the algorithm is linear, but because of the predictions, there is some form of optimization that is done, making the times better. Looking at the Round Robin, we can tell that the increase in time is definitely not linear, but logarithmic. But because it is not optimized, the logarithmic time is not optimal, but still better than the rest of the other algorithms.

In the end I would say that the experimentation and development was interesting yet difficult. It was interesting because the topic and the algorithms I have covered are very interesting to me and I think that they are a valuable way to learn and to understand the workings of the operating system. But there are also a lot of difficulties that were involved. Writing this project in C++ cause a lot of problems, many lines of code, and a lot of implementation details that are difficult to figure out. Because of this, valuable time is lost on fixing problems that are not concerned with the actual heart of the problem.

I think that I proved what I set out to do and to compare how the different procedures interact and how they work. There is plenty of work to be done more, like writing more detailed algorithms and comparing them on a more linear system, without having to visualize them.

7 References

- [1] <https://en.wikipedia.org/wiki/Red>
- [2] <https://www.linuxjournal.com/node/10267>
- [3] Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). Operating system concepts. Hoboken, N.J: Wiley.
- [4] <https://www.coders-hub.com/2015/07/red-black-tree-rb-tree-using-c.html>
- [5] <https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/>
- [6] <https://www.mjmwired.net/kernel/Documentation/scheduler/sched-design-CFS.txt>
- [7] https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- [8] <https://developer.ibm.com/tutorials/l-completely-fair-scheduler/>
- [9] <https://www.cs.usfca.edu/~mmalensek/cs326/schedule/lectures/326-L7.pdf>
- [10] <https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>
- [11] <https://oakbytes.wordpress.com/2012/07/03/linux-scheduler-cfs-and-virtual-run-time/>
- [12] <https://stackoverflow.com/questions/54565683/at-what-point-does-a-program-become-a-process-virtual-machine>
- [13] <https://stackoverflow.com/questions/19181834/what-is-the-concept-of-vruntime-in-cfs>