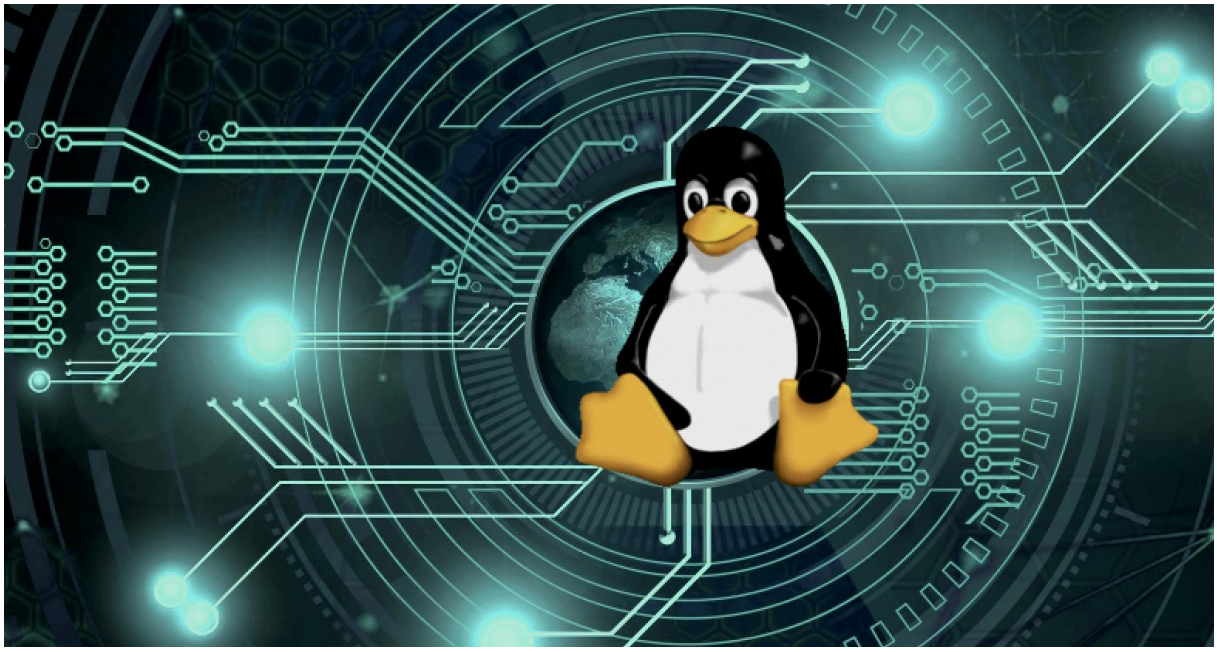
Assignement 1 Report : CPU Scheduler Simulation



Professor :

- Dr. Youssef Iraqi

BY :

- Anass Kemmoune
- Mahmoud Maftah

Table of Content

1-Introduction

This report delves into the realm of CPU scheduling algorithms, utilizing a simulation program to investigate their behavior and impact on system performance within operating systems. The assignment entails the implementation of five distinct scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, Round Robin (RR), and a combination of Priority Scheduling with RR. (We chose to add a sixth algorithm which is Round Robin with aging)

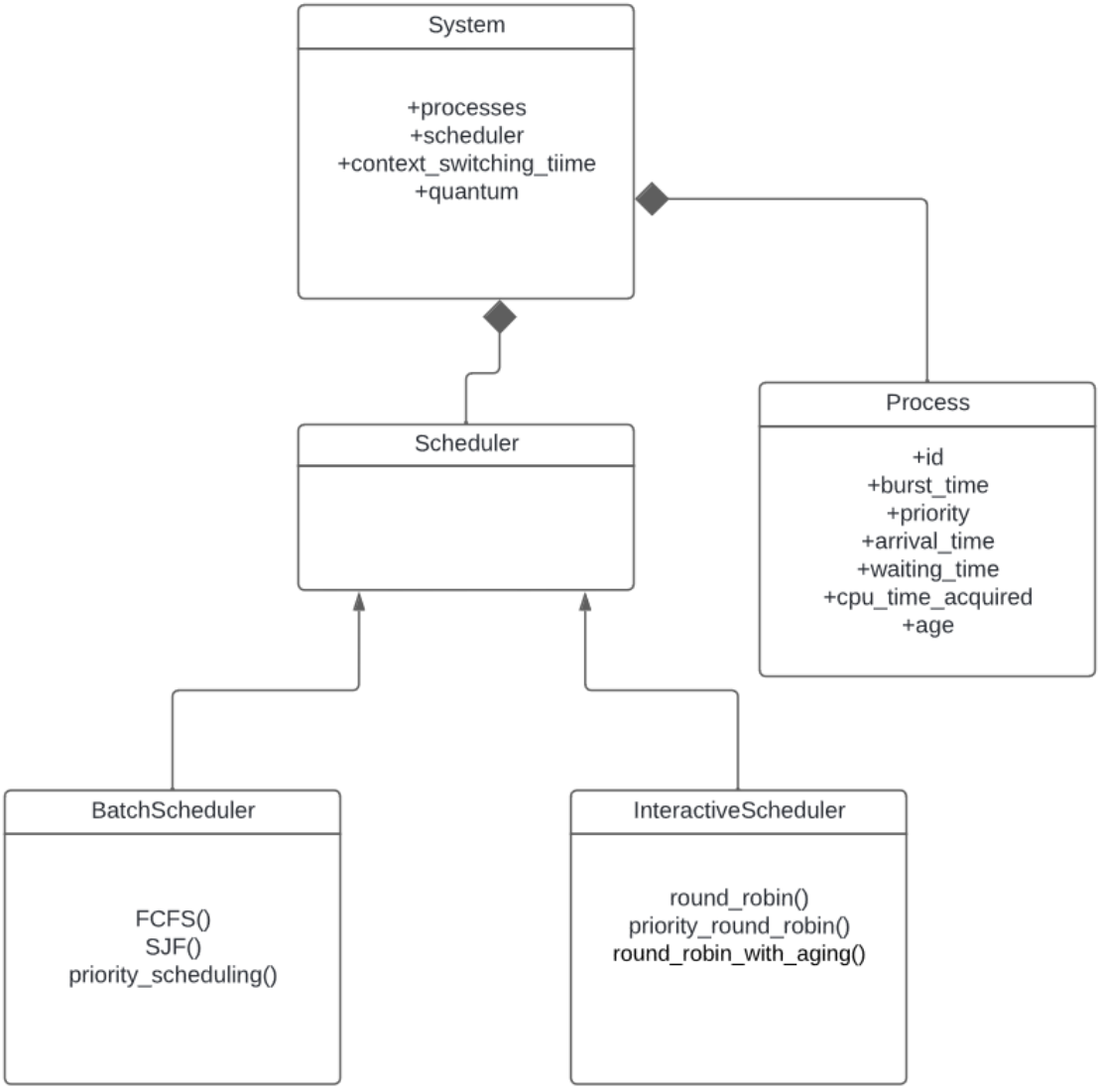
2-Interactive Scheduling algorithms comparison:

1. *Round Robin (RR):*
    - **Principle**: Each process is assigned a fixed time slice or quantum, and the CPU scheduler executes them in a circular queue fashion. Once a process's time quantum expires, it's moved to the end of the ready queue.
    - **Advantages**:
        - Fairness: Ensures all processes get a chance to execute within their time slice.
        - Suitable for time-sharing systems, ensuring responsiveness.
    - **Disadvantages**:
        - Higher overhead due to context switching at each time slice.
        - Not suitable for tasks requiring strict deadlines or real-time constraints.
2. *Priority Round Robin (PRR):*
    - **Principle**: Similar to Round Robin, but each process is assigned a priority. The scheduler selects the highest priority process and executes it for a fixed time slice. If a higher priority process arrives during the execution of a lower priority one, it preempts the lower priority process.
    - **Advantages**:
        - Allows for prioritization of tasks, ensuring critical processes are executed promptly.
        - Maintains fairness by incorporating round-robin principles.
    - **Disadvantages**:
        - May still suffer from priority inversion issues.
        - Lower priority processes can experience starvation if higher priority ones keep arriving.
3. *Round Robin with Aging:*
    - **Principle**: A variant of Round Robin where the priority of a process increases with its waiting time. As a process waits in the ready queue, its priority gradually increases, ensuring that long-waiting processes eventually get scheduled.

- **Advantages**:
    - Mitigates the problem of starvation by dynamically adjusting priorities.
    - Maintains fairness while also addressing the issue of aging processes.
- **Disadvantages**:
    - Complexity in implementing aging mechanisms.
    - Requires careful tuning of aging parameters to avoid unintended consequences such as priority inversion or excessive aging.

3-Design Choices

We have decided to go for the following architecture in order to mimic the real representation of an operating system and we also wanted to make it more suitable for the next assignment:



4-Implementation details/ data structure choices and time complexity analysis:

In this section we will go through each algorithm separately explaining how fast it goes, it's asymptotic behavior when the number of processes gets large and most importantly our implementation choices especially when it comes to data structures.

Scheduling Algorithms for Batch Jobs:

- First come First Served (FCFS):

This is the simplest of all algorithms that will be presented, we simply sort the list of processes we get as input based on their arrival time (In case of tie we simply refer to the ID, this is not part of the algorithm itself but it's just a convenient way to pick a random among processes that arrived at that time.) and we keep updating a global variable we maintain, each time we finish a process we automatically select the next one from our sorted array update its parameters and then move on.

1. **Sort incoming jobs by their arrival time**. If there are ties in arrival time, resolve them by referring to the job ID.
2. **Initialize a global variable to keep track of the current time**. This variable represents the time elapsed since the start of execution.
3. **Select the first job from the sorted list**.
4. **Execute the selected job until completion**. Update the global time variable to reflect the time taken by the job to complete.
5. **Once a job is completed first load into queue all jobs that have arrived, only then select the next job from the sorted list**. If multiple jobs have the same arrival time, select them in the order of their IDs.
6. **Repeat steps 4-5 until all jobs are completed**.

Time Complexity: In this algorithm we are simply using an Array and sorting jobs so the algorithm needs O(n*log(n)) for sorting and an additional O(n) to derive information related to each job.

The resulting complexity is **O(n*log(n))**

- Shortest job first (SJF):

In this algorithm, we first sort incoming jobs by their arrival time. Then, we maintain a global variable called '**Current time'** and a priority queue where jobs are prioritized by their burst time (time required for execution). We repeatedly select the top element from the queue (a priority queue always returns the shortest job among all those it contains), execute it, and update its attributes. Afterwards, '**current time'** increases by the burst time of the process that was just executed. Before proceeding, we should load into

the queue all jobs that have arrived by the time we were executing the previous process, and only then select the one with the lowest burst time for the next run.

Time complexity: we decided to use a priority queue for this algorithm as we frequently need to perform update/ get queries which can be carried out easily by a heap (priority queue) in **log(n)** time where n is the number of instances in our queue then each job is inserted into the heap exactly once with cost log(n) and removed exactly once with cost **log(n)** we can use the total number of processes as a great upper bound for 'n' as we can't have prior knowledge about the number of elements in the heap for every possible combination of jobs. This would yield an additional O(n*log(n)) complexity.

- The resulting Complexity is **O(n*log(n))**

Priority Scheduling:

In Priority Scheduling, jobs are sorted based on their priority instead of their burst time. This means that the process with the highest priority is selected for execution first. Just like Shortest Job First (SJF), we maintain a global variable for 'Current time' and use a priority queue to prioritize jobs.

The difference lies in how jobs are prioritized. In Priority Scheduling, jobs are sorted based on their priority level, which could be predefined or dynamically assigned based on factors like importance, deadline, or resource requirements.

The algorithm follows a similar approach to SJF:

1. Sort incoming jobs by their arrival time.
2. Maintain a priority queue where jobs are prioritized by their priority level.
3. Select the top element from the queue (highest priority job), execute it, and update its attributes.
4. Increase 'current time' by the execution time of the selected process.
5. Load into the queue all jobs that have arrived by the time the previous process was executed.
6. Select the highest priority job for the next run.

The time complexity of Priority Scheduling is **O(n*log(n))** by using the same reasoning as the one above.

Scheduling algorithms for interactive systems:

- Round Robin:

Similar to previous algorithms we first sort jobs by arrival time, maintain a queue (or deque in python) which is a data structure that supports insertions and deletions in constant time, and we keep picking the job in the front, run it for a quantum of time, if it is done we don't return it, otherwise we add all jobs that have arrived by the time it was running and only then add it.

1. The steps for Round Robin scheduling are as follows:
2. Sort incoming jobs by their arrival time.
3. Maintain a queue data structure to store jobs.
4. Select a job from the front of the queue and execute it for a fixed quantum of time.
5. If the job completes within the quantum, it is removed from the queue.
6. If the job doesn't complete within the quantum, it is placed back in the queue.
7. After each quantum, check for newly arrived jobs and add them to the queue.
8. Continue this process until all jobs are completed.

Time Complexity: The time complexity of Round Robin scheduling is **O(n*k),** where 'n' represents the number of processes and k is the expected number of times we will come across some job we can easily compute it for some process as **K = (burst time/ quantum).**

- Priority Round Robin:

Priority Round Robin scheduling follows similar steps to Round Robin, but it additionally considers the priority of jobs. These are the steps for the Round Robin algorithm for Priority Round Robin scheduling:

1. **Sort incoming jobs by their arrival time and priority**.
2. **Maintain a priority queue (min heap) to store jobs**. Jobs with higher priority should be at the front of the queue. Each job in the queue should have its remaining burst time and other necessary information.
3. **Select a job from the front of the queue and execute it for a fixed quantum of time**.
4. **If the job completes within the quantum, it is removed from the queue**.
5. **If the job doesn't complete within the quantum, it is placed back in the queue**.
6. **After each quantum, check for newly arrived jobs and add them to the queue**. If a newly arrived job has higher priority than the job currently running, preempt the running job and execute the higher priority job.
7. **Continue this process until all jobs are completed**.

Time Complexity: for this Algorithm finding the time complexity might be a little bit trickier, for each running process we will need to perform insertions using a heap as an underlaying data structure, in **O(log(n))** time, this operation will be done a total of K = (burst time/ quantum) therefore the expected time complexity will be **O(n*log(n)*k)** where K is the factor we defined above.

- Round Robin with Aging

Using the approach where the age of a process increases upon execution, is implemented as follows:

1. **Sort incoming jobs by their arrival time**.
2. **Maintain a heap (or priority queue) to store jobs**. Each job in the queue should have its remaining burst time and other necessary information.
3. **Select a job from the front of the queue and execute it for a fixed quantum of time**.
4. **If the job completes within the quantum, it is removed from the queue**.
5. **If the job doesn't complete within the quantum, it is placed back in the queue**.
6. **After each quantum, check for newly arrived jobs and add them to the queue**.
7. **While adding newly arrived jobs to the queue, update their age**. Age is a variable associated with each job indicating the time it has spent waiting in the queue without being selected for execution. In this case, whenever a job is executed, its age is incremented.
8. **Select the next job for execution based on its age**. The job with the lowest age is chosen for execution next. This ensures that jobs that have been waiting longer are given priority.
9. **Continue this process until all jobs are completed**.

By increasing the age of a process upon execution and selecting the process with the lowest age for the next run, Round Robin with Aging prevents starvation by prioritizing jobs that have been waiting in the queue for a longer time.

In terms of time complexity, the main overhead is still in the constant-time operations of adding, removing, and updating jobs in the queue done in **log(n)**, along with the additional operation of updating the age of a process upon execution. The time complexity remains like standard Round Robin scheduling, with the aging mechanism adding a minor overhead.

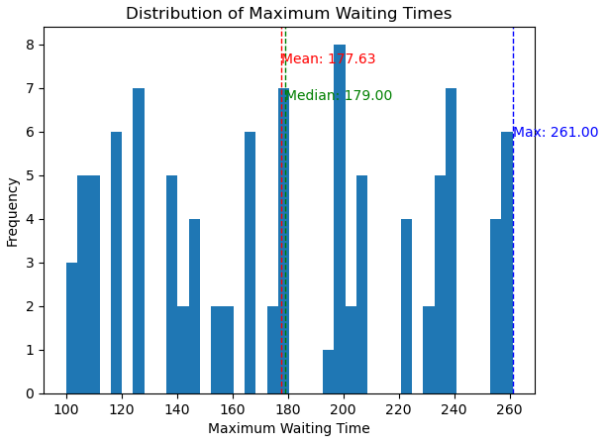
5-Limitations of Round Robin and introduction to Aging:

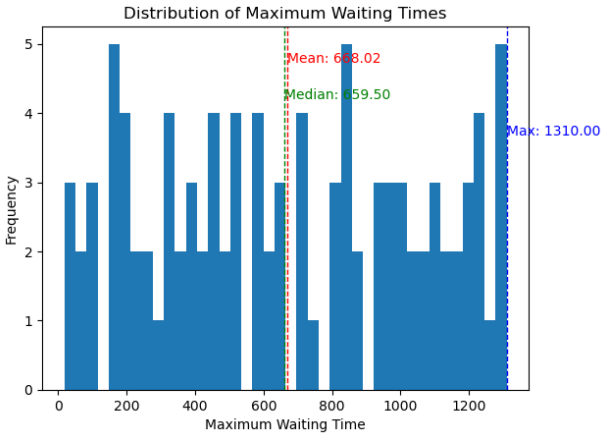
While the Priority Round Robin algorithm is commonly employed in interactive systems prioritizing response time, it is crucial to acknowledge its significant drawbacks. Consider a scenario where a single computer is shared among multiple users, with the administrator holding the highest priority, allowing them to run processes with top precedence. In the typical round robin algorithm, normal user processes are only executed when there are no administrator processes in the queue. However, in an interactive system, allocating even a few milliseconds to normal users can substantially enhance overall performance. Here lies the flaw in the traditional round robin algorithm's approach, as it fails to adequately address the importance of aging and responsiveness for all users in such shared environments.

The proposed solution to address the shortcomings of the round robin algorithm in shared computing environments is the implementation of the Priority Round Robin algorithm with aging. This enhanced algorithm, as conceptualized by our team, incorporates an additional parameter per process known as "age." The age of a process determines its likelihood of execution, with younger processes being prioritized.

In the Linux operating system, age updates are typically managed at the hardware level, resulting in more efficient execution times. However, in our customized version, we dynamically update the age of each process every time it runs, based on its priority. This means that processes with higher priority experience slower decreases in their age, ensuring they retain a higher likelihood of execution very soon.

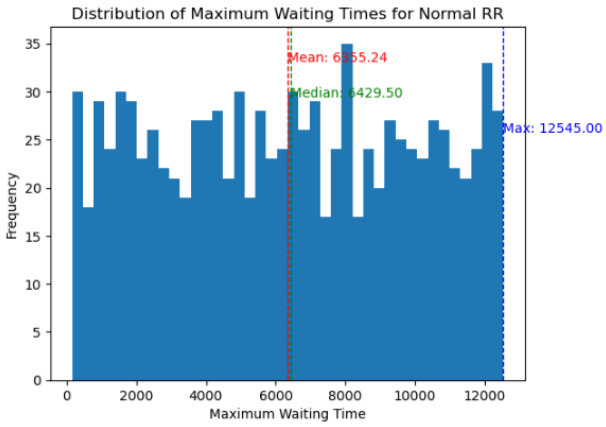
In this Test We will generate a list of random jobs and display how the aging feature prevents processes from starving (experiencing large periods of time without CPU):

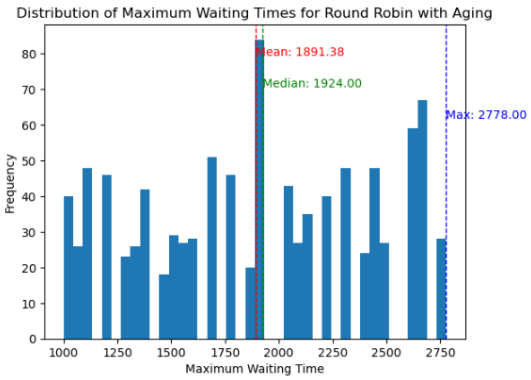




As we can see Round Robin With aging clearly outperforms the normal RR algorithm, as it only yields **261ms** in the worst case as opposed to the **1310ms** for normal RR.

We have run the same algorithms on a larger dataset, the results are pretty much the same: the distribution of maximum values for waiting times captures the entire difference in performance between the two algorithms.

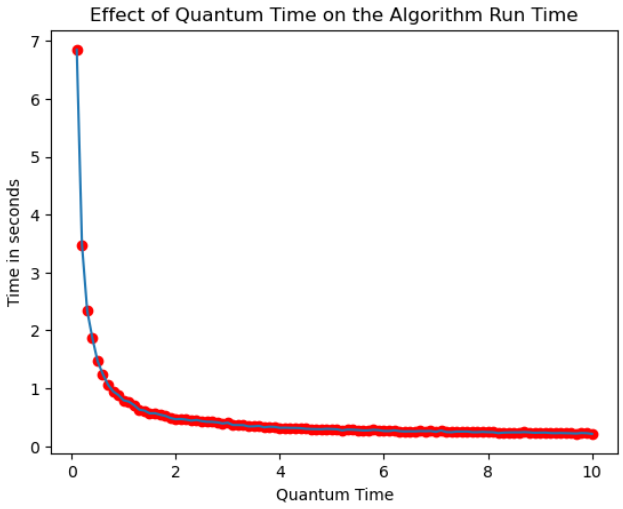


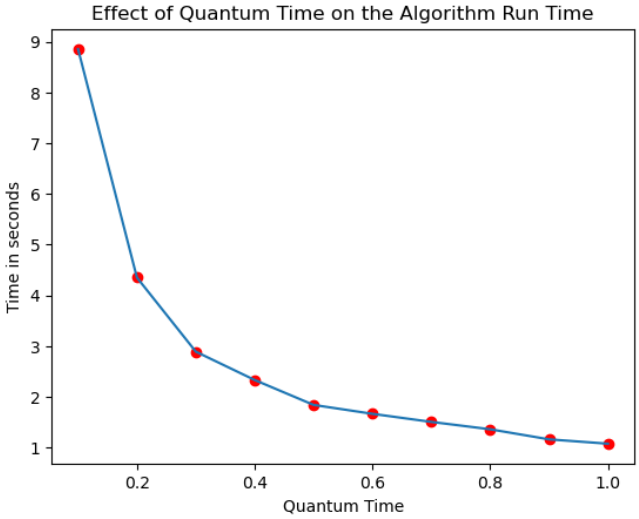


The code for this testing part can be found in the deliverable as **AgingTests.ipynb**

6-Benchmarking

- Effect of the Quantum time on the algorithm runtime:



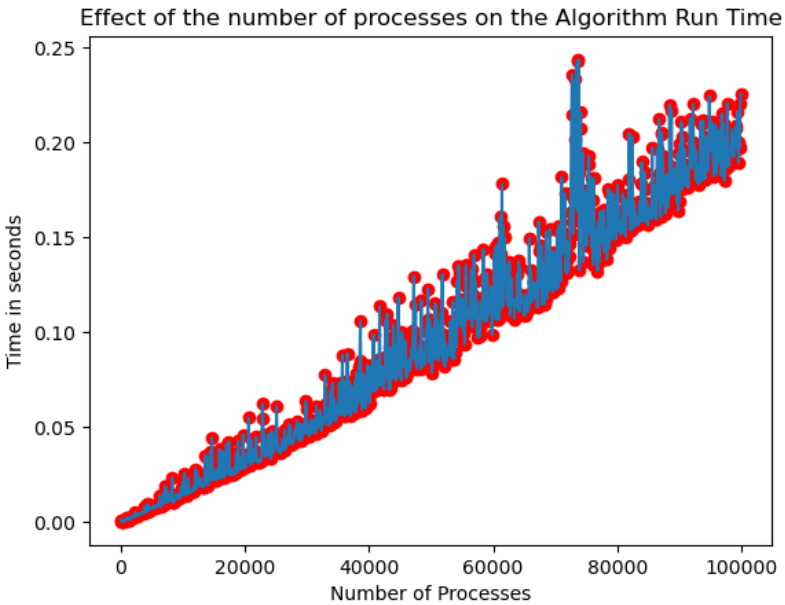


These pictures illustrate how the runtime of our preemptive algorithms is influenced by quantum time. As previously demonstrated it grows in O(1/q), where q represents the quantum duration. This behavior is evident in the graph, which depicts the average running time of a sequence of Round-Robin (RR) algorithms with only the quantum as the differing factor, run on a wide range of randomly generated jobs.

You can have access to the entire Benchmark testing code in the file (Benchmark.ipynb) attached to the deliverable.

- Effect of the number of processes:

We might think that the Processes count doesn't affect the scheduling algorithm performance and that only total burst time matters, but the thing is that the overhead of keeping lots of processes simultaneously in the queue will affect the log(n) factor previously mentioned.



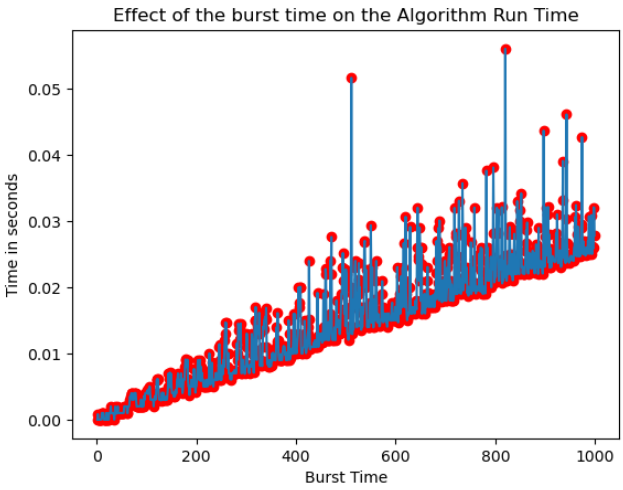
- Effect of the burst time on the algorithm Run time.

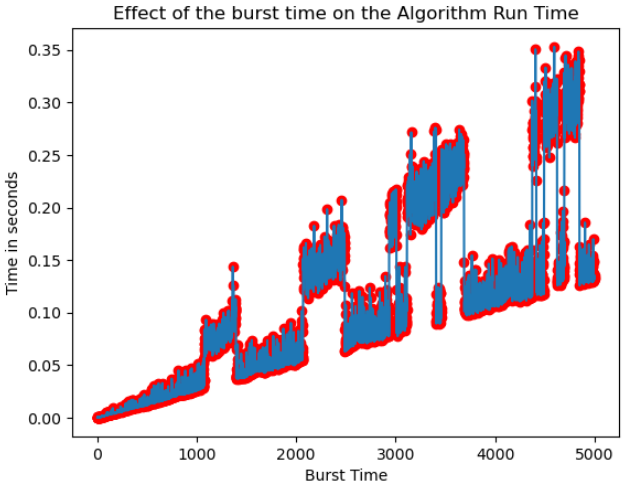
To capture the exact information, we will use 1s as quantum, this way the difference between a process with burst time 5 and burst time 10 will be fully kept as opposed to using 5s as a quantum.

- ✓ Maximal arrival time = 100
- ✓ Context switching time = 0
- ✓ Quantum = 1
- ✓ Number of processes = 100

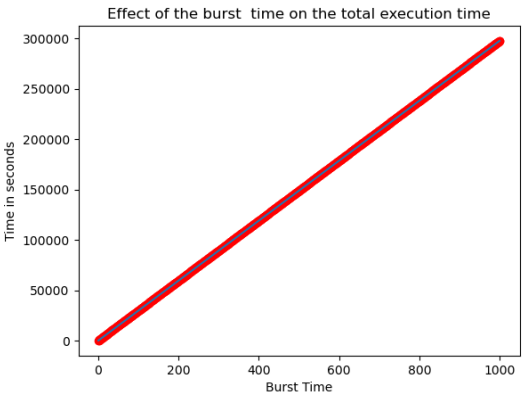
And we will test our algorithm on the following range of burst times

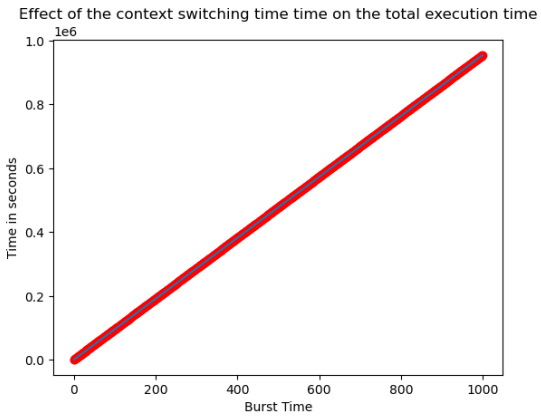
➔ Burst time range = range(1, 5000)





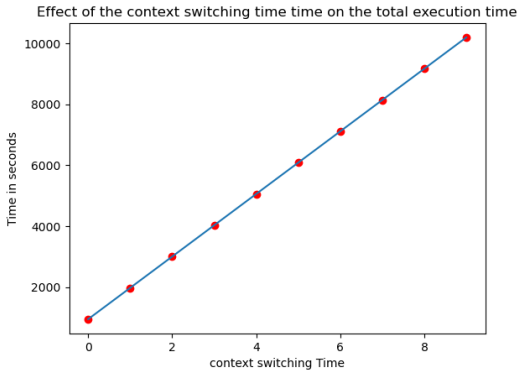
- Benchmark test on time taken to complete running all processes (Not for running the actual scheduling algorithm)

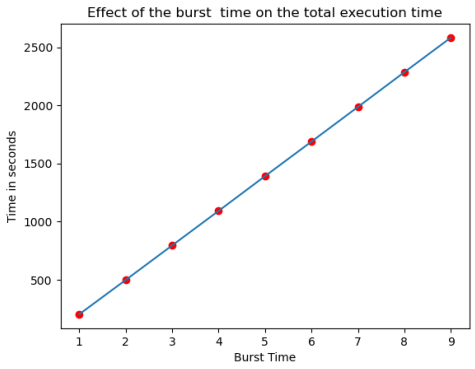




As we can see if we do not include time needed for managing processes we get a clearly linear curve without errors.

To capture more the effect of burst time and context switching overhead, we will now use a smaller number of processes and compare the effect on the total execution time.





Note: You can find the code for our benchmark tests in **Benchmarks/Benchmarks.ipynb**

Extensive Benchmark tests for batch jobs

In this section we will compare how the three different algorithms we have for batch systems namely: **Priority Scheduling**, **First come first served** and **shortest job first** perform on a large set of randomly generated processes, we will use the same testing set consisting of **1e6** different jobs with burst time ranging from 5 to 20 seconds, note that the burst time in this case wont have any effect on the benchmark test, indeed it will only make the plots larger which would be of no added value.

We will use this testing confirm information seen in class:

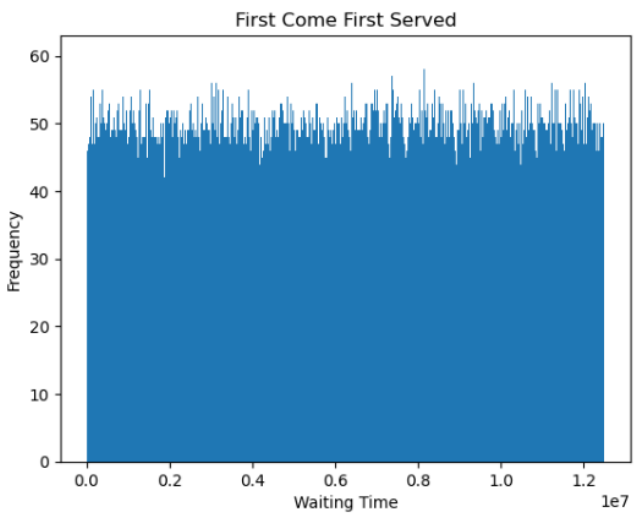
“Shortest job first has the best possible average turnaround time for a set of processes that arrives to the system at the same time (say time 0).”

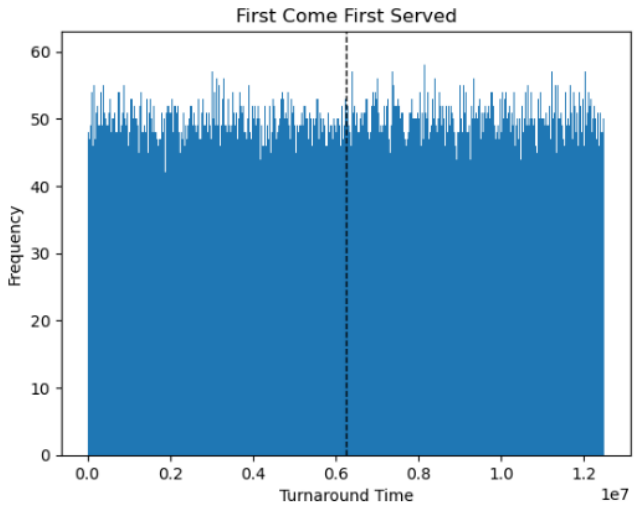
Pipeline:

1. **Implementation:** Each scheduling algorithm (Priority Scheduling, FCFS, and SJF) will be implemented according to their respective principles and logic.
2. **Execution:** The 1 million processes will be sequentially scheduled using each algorithm. The scheduling decisions and corresponding turnaround times will be recorded for analysis.
3. **Metrics:** The following metrics will be evaluated for each algorithm:
    - **Average Turnaround Time:** This metric reflects the average time taken for a process to complete execution from the moment it enters the system.
    - **Throughput:** The throughput measures the number of processes completed per unit of time, providing insights into the system's efficiency.

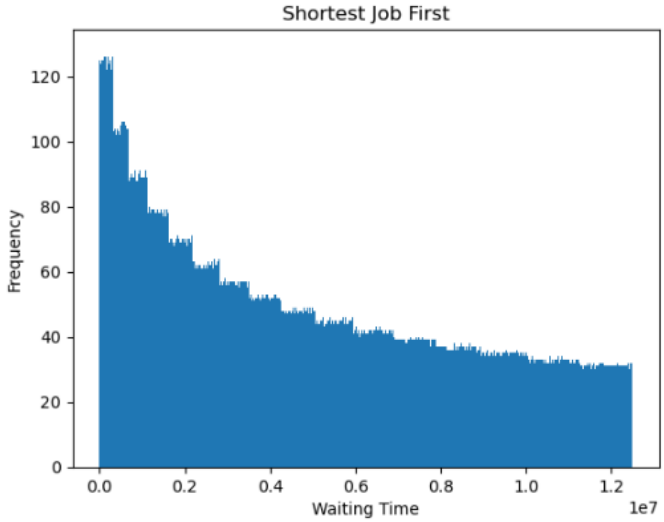
4. **Analysis:** The collected data will be analyzed to compare the performance of each algorithm concerning the defined metrics. Special attention will be given to SJF to assess its efficacy in minimizing turnaround times, particularly for simultaneous process arrivals.

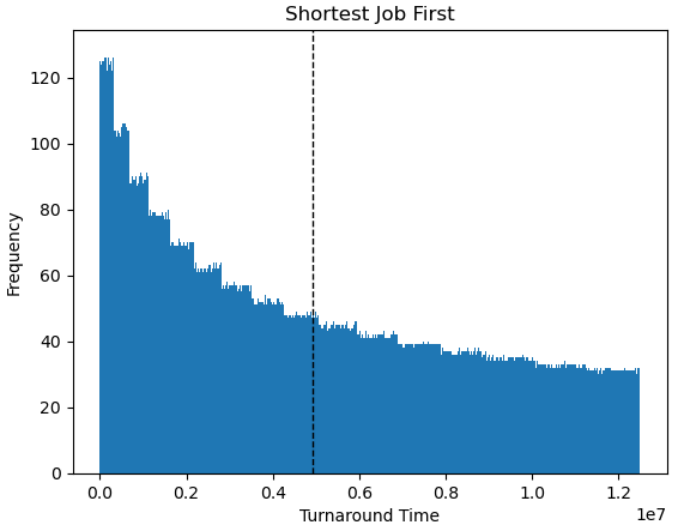
- First come first served.



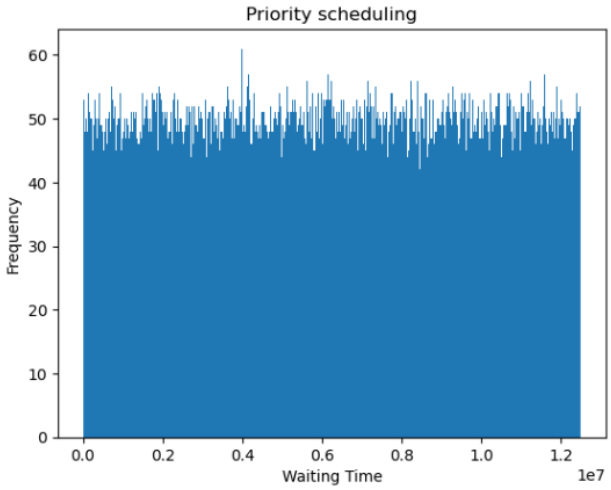


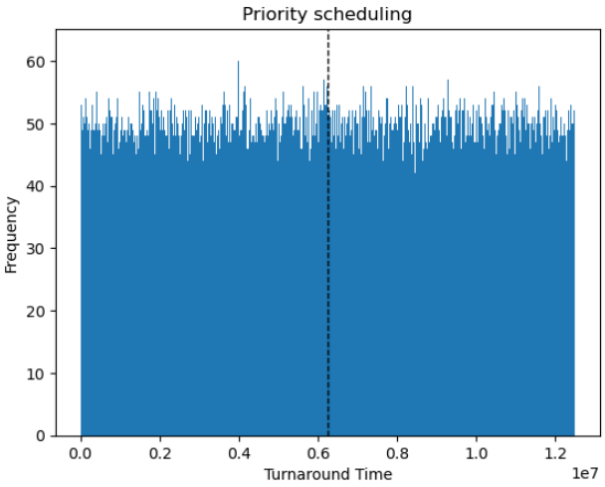
- Shortest Job First





- Priority scheduling





7-

Code Testing

For the code testing part, we have created a separate file (**Testing.ipynb**) that shows how each of the six scheduling algorithms we have implemented can run on a randomly generated dataset of jobs. The notebook contains code snippets demonstrating the execution of each scheduling algorithm on the dataset. You can either execute the notebook on your end or simply review the results that we've already computed.

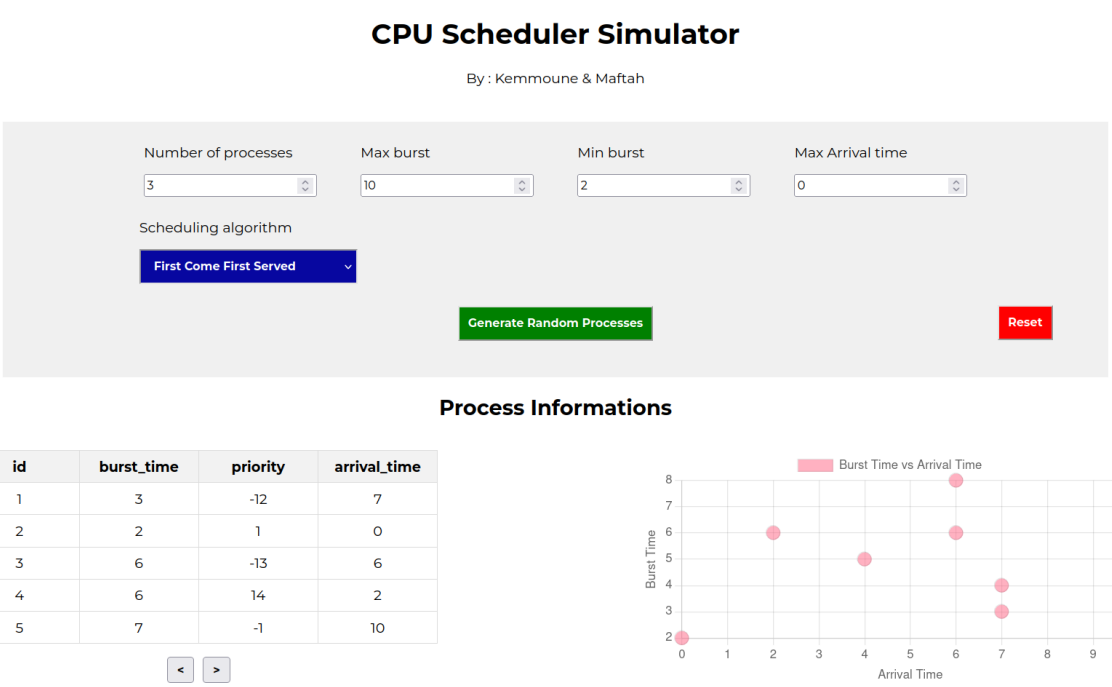
For each algorithm we load the previously generated csv file and then run it on the set of jobs, afterwards we simply print the **average turnaround time** (**for batch systems lowest value is held by SJF algorithm**), the **average waiting time**, and finally the **average maximum waiting time**, by maximum waiting time we refer to the longest burst of time a process have gone through without having CPU this value can get quite large especially with Priority Algorithms, and one more time we can validate a remark we could make in class **'the RR with aging detains the best value with only 3000ms compared to 7000ms for normal RR algorithm'** (with and without Priority).

More and more details are to be found in the (**Testing.ipynb**) file attached to the deliverable.

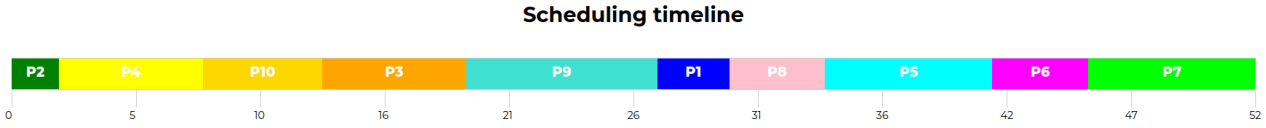
8-Graphical Interface and Visualization

For the graphical Interface, it was done using HTML CSS JS, and linked with the backend through Python's Flask.

It includes a Form where the user can fill the range of the process parameters to generate random processes.



After the user input, the interface shows the process table and a scatter plot showing the distribution of the processes depending on burst time and arrival time



Output Table

id	burst_time	priority	arrival_time	turnaround_time	waiting_time
1	3	-12	7	23	20
2	2	1	0	2	0
3	6	-13	6	13	7
4	6	14	2	6	0
5	7	-1	10	31	24
6	4	19	10	35	31
7	7	-7	10	42	35
8	4	10	7	27	23
9	8	16	6	21	13
10	5	-12	4	9	4

Output Statics

Average Turnaround Time :
19

And then the scheduling algorithms runs the processes, and the GUI shows a Gantt diagram for the whole execution pipeline in addition to an output Table. Furthermore, it shows more statistics and graphical representations as shown below.

Output Statics

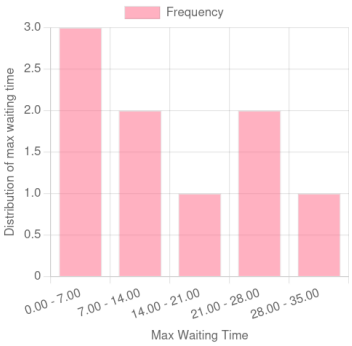
Average Turnaround Time :
19

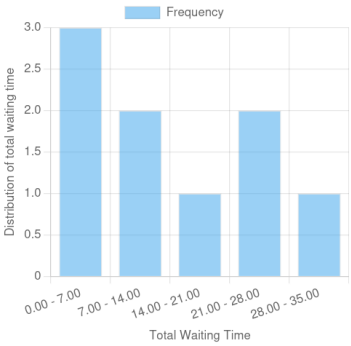
Average Waiting Time :
14.272727272727273

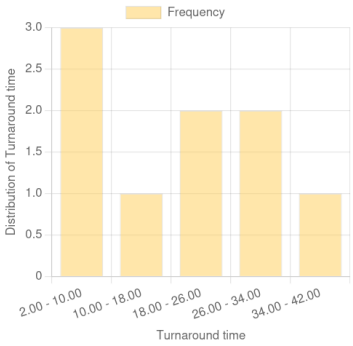
CPU utilization :
100%

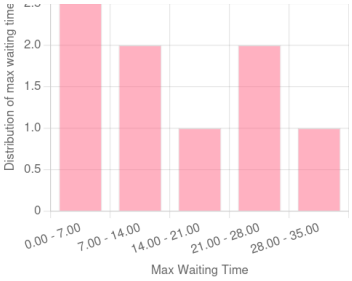
Output visualization

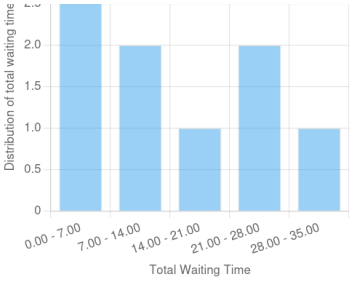
Time Distribution Chart

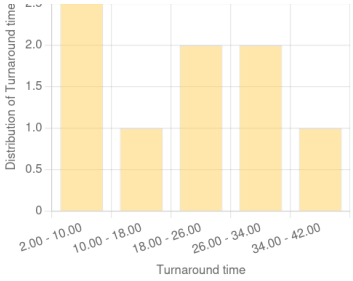












CPU utilization Chart