Optimizing CPU Process Scheduling in Operating Systems: Utilizing Binary Trees as Min Heaps in the Priority Scheduling Algorithm

Muhammad Rasheed Qais Tandjung - 13522158
Department of Informatics Engineering
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Jl. Ganesha 10 Bandung 40132, Indonesia
13522158@std.stei.itb.ac.id

***Abstract*—CPU scheduling is a core component in operating systems, in which a scheduling algorithm is used to determine the process to be executed next in a queue. The priority scheduling algorithm is an example of such algorithm that uses priority queues to determine the next executed process. As opposed to the traditional use of a linear priority queue for this algorithm, this paper proposes the use of min-heaps, a type of priority queue implemented as a binary tree. With the utilization of binary trees, the time complexity of insertion and deletion to the queue can be heavily reduced from linear time to logarithmic time, thus maximizing CPU performance.**

***Keywords*—Binary Trees, CPU Scheduling, Data Structures, Heaps, Priority Queues**

I. INTRODUCTION

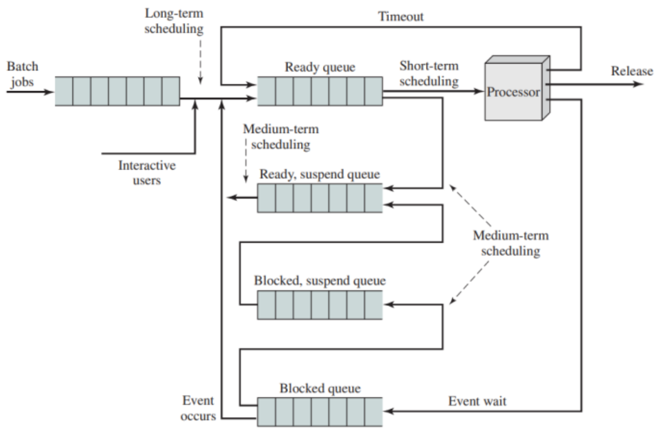


Figure 1.1 An illustrated overview of CPU scheduling.
Source: [3]

In the realm of operating systems, the efficient allocation of CPU resources is vital to ensure optimal system performance. The CPU, following its name, is the central location for executing every process given by the operating system. However, its ability to handle tasks concurrently is limited by the number of physical cores it possesses.

Despite this limitation, however, in reality the usual amount of processes needing to be executed at a certain moment far exceeds the capacity that the CPU can handle. Therefore, to counteract this problem, the operating system implements what is called *process scheduling* to determine which processes should be executed first.

Process scheduling refers to the mechanism through which the operating system manages the execution of multiple processes. When numerous processes are queueing for CPU time, the scheduler determines the order in which these processes are executed, aiming to maximize CPU utilization, minimize response time, and enhance overall system throughput.

At its core, process scheduling involves various algorithms that dictate how the CPU selects the next process to execute from the ready queue. One of these algorithms, aptly called *priority scheduling*, operates on the principle of assigning priorities to different processes based on specific criteria. Processes with higher priority are granted access to the CPU before those with lower priority.

These processes are stored in a data structure called *priority queues*, where the processes in front of the queue gets executed first, and when more processes get added, they are inserted to the back of the queue. Processes with higher priority will be pushed front to its appropriate location in the queue by the operating system.

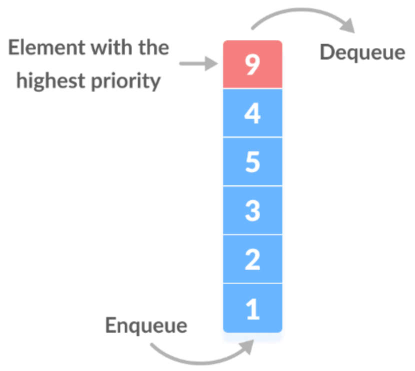


Figure 1.2 A traditional linear priority queue. These data structures suffer from O(n) insertion time.
Source: [4]

Traditionally, priority queues have been implemented using linear data structures such as linked lists or arrays. However, these structures suffer from limitations in terms of efficiency and scalability. As the number of processes increases, searching for the highest-priority process becomes computationally expensive, hindering overall system performance.

To address these limitations, this paper proposes the utilization of binary trees as heaps in the implementation of the priority scheduling algorithm. Heaps offer a compelling alternative due to their inherent properties: they guarantee efficient insertion, deletion, and retrieval of the highest-priority element, all with a logarithmic time complexity. This significantly improves the efficiency of the scheduling process, particularly for large and dynamic workloads.

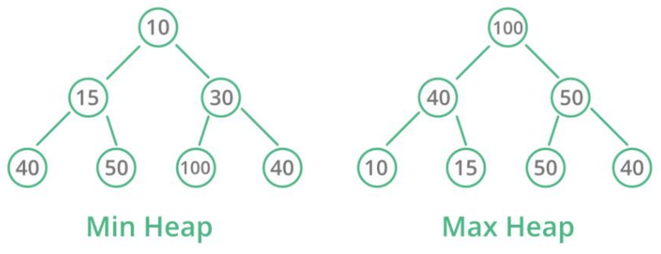


Figure 1.3 The heap data structure improves on linear data structures by having O(log n) time operations.
Source: [5]

II. Theoretical Framework

A. Trees

In graph theory, trees represent a specific class of graphs characterized by unique properties that make them invaluable for modeling and analyzing various real-world phenomena.

The concept of a tree is inherently intuitive. It evokes images of branching structures, with a single root node connected to multiple child nodes, which can further branch out to form subsequent levels. This hierarchical organization allows for efficient traversal and navigation, making trees ideal for representing hierarchical relationships and data structures.

Formalizing the intuitive notion of a tree requires precise mathematical language. A tree can be defined as a connected, acyclic, and undirected graph [1]. This definition captures several key properties:

1. Connected: All nodes in the tree are reachable from every other node via a sequence of edges. This interconnectedness ensures that the tree functions as a single cohesive unit.

2. Acyclic: The tree does not contain any cycles. This means that traversing the edges in any direction will not lead you back to the same node you started from. Acyclicity prevents infinite loops and guarantees a well-defined structure.

3. Undirected: The edges in a tree have no inherent direction. This means that the relationship between nodes is symmetrical, and you can traverse the edges in either direction without changing the meaning of the connection.

These three conditions are sufficient to characterize a mathematical tree. Figure 2.1 illustrates the distinction between trees and regular graphs (non-trees).

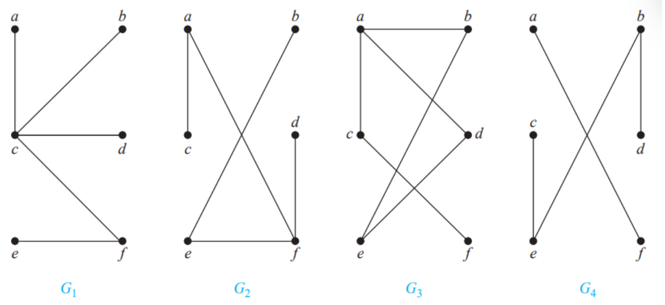


Figure 2.1 Illustration of trees and non-trees
Source: [1]

Graphs G_1 and G_2 in Figure 2.1 fit the criteria of being connected, acyclic, and undirected, and therefore can be considered trees. Graph G_3 cannot be classified as a tree, since there exists a circuit *a-b-e-d-a* in the graph, violating the acyclic criteria of trees. Graph G_4 is not a tree either, since it consists of two unconnected graphs.

The distinction between trees and graphs consist purely of the three criteria previously mentioned. In practice, however, it is much more common for a different type of tree to be used instead. These trees are called *rooted trees*, and applications of trees in various fields see far more use of this type of tree instead of the mathematical tree described above.

B. Rooted Trees

Rooted trees are a specific subclass of trees characterized by a unique root node. This root serves as the starting point for navigating the entire tree structure.

Unlike other trees, where all nodes are treated equally, rooted trees establish a clear hierarchy and directionality. This inherent organization facilitates efficient searching, data retrieval, and manipulation, making them invaluable tools for representing complex systems and data sets.

Formally, a rooted tree can be defined as a connected, acyclic, and directed graph with the following properties:

1. Connected: All nodes in the tree are reachable from every other node via a sequence of edges. This interconnectedness ensures that the tree functions as a single cohesive unit.

2. Acyclic: The tree does not contain any cycles. This means that traversing the edges in any direction will not lead you back to the same node you started from. Acyclicity prevents infinite loops and guarantees a well-defined structure.

3. Directed: Each edge in the tree has a specific direction, pointing from a parent node to a child node. This directionality establishes a clear hierarchy and allows for efficient navigation through the tree structure.

4. Unique Root: There exists a single node, designated as the root, with no incoming edges. This node serves as the starting point for traversing the entire tree.

With the addition of direction and hierarchy in rooted trees, several new terminologies are introduced to describe trees, the nodes inside trees, and relations between nodes inside the tree:

Parent: A node connected to a child node by a single directed edge.

Child: A node connected to a parent node by a single directed edge.

Siblings: Nodes that share the same parent node.

Root: The node at the topmost of the tree with no parent node.

Degree: The number of outgoing edges connected to a node.

Level: The depth of a node in the tree, with the root being at level 0.

Subtree: A rooted tree formed by a node and all its descendants.

Depth: The length of the longest path from a node to the root node.

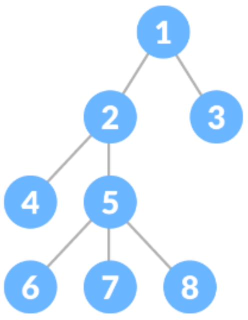


Figure 2.2 Illustration of a rooted tree, with node (1) as its root, and the subtrees (2) and (3) as its children.
Source: [2]

With a wide range of applications for the rooted tree data structure, many seek to invent further derivations of this concept, with some seeing even more uses in various problems spanning all sorts of fields.

One such derivation is the N-ary tree, where each node can only have a maximum of N children, however having less children is still allowed. While this type of tree can still see various use cases, by far the most significant role of the N-ary tree in the realm of data structures is not the N-ary tree itself, but yet another derivation of the N-ary tree, also known as the *binary tree*.

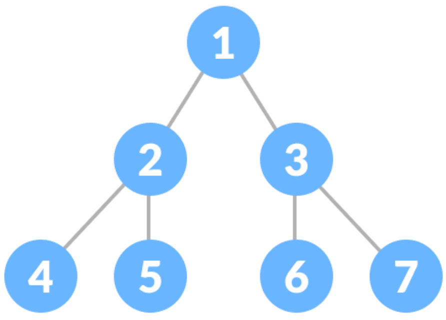


Figure 2.3 The binary tree, one of the most important data structures in computer science.
Source: [6]

The binary tree is a special type of N-ary tree where each node can have a maximum of two children. While this choice of restriction seems oddly arbitrary, the binary tree is a structure that surprisingly has an extraordinary amount of applications, especially in the computer science field. Some of its main uses include, but not limited to: binary search trees (BSTs) for efficient storing of sorted data, Huffman coding for data compression, and the main topic of this paper, *the binary heap*.

C. Binary Heaps

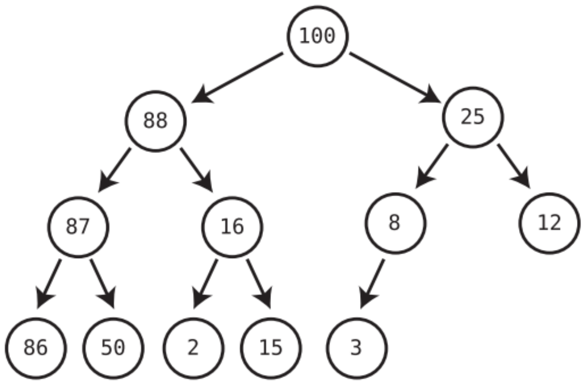


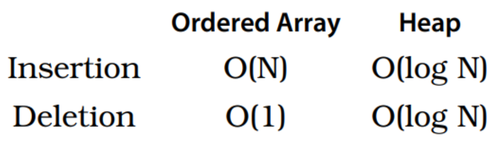
Figure 2.4 Illustration of a max-heap
Source: [7]

A binary heap is a specific kind of binary tree that satisfies two conditions [7]:

1. The binary tree is nearly complete, which means empty nodes are only in the lowest layer, and the insertion is always to the leftmost empty slot.
2. Every node in the tree is always (1) less than, or (2) greater than both of its children. A heap that satisfies condition (1) is called a *min-heap,* while a heap that satisfies condition (2) is called a *max-heap*.

Unlike other tree data structures, the heap doesn't support searching or random deletion, and its main operations are delete root and insertion, making it quite similar to the queue.

To insert into a node into the heap, we first insert it to the leftmost node in the bottom layer, and "bubble up" the node to its correct position. To delete the root from the tree, we first swap the root with the last node in the lowest layer, delete the last node, and then the current root (which was previously the last node) is "bubbled down" to its correct position.



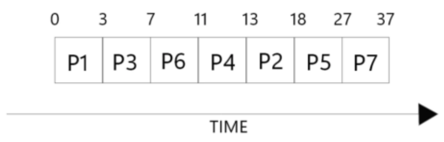
	Ordered Array	Heap
Insertion	O(N)	O(log N)
Deletion	O(1)	O(log N)

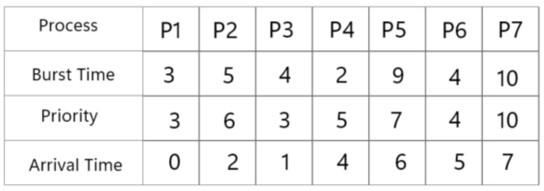
Figure 2.5 Time complexity comparison of ordered array versus heaps
Source: [7]

This makes the steps in insertion and deletion always less than or equal to the root height (log n). Another unique characteristic of the heap is that the root is always either the maximum or the minimum value of the entire tree. If we imagine the numbers stored in a heap as a "priority value", then the root node will always store the node with maximum priority. These two characteristics of the heap is the reason why it is an excellent choice to be used as a priority queue.

D. CPU Scheduling

CPU scheduling is a pivotal aspect of operating systems, playing a fundamental role in optimizing the utilization of the CPU and enhancing overall system performance. It encompasses the methodology by which the system selects and allocates available CPU resources to various processes. Through the intricate orchestration of competing demands and priorities among multiple processes, CPU scheduling seeks to achieve optimal throughput, minimize response times, and ensure fair access to system resources, thereby contributing significantly to the seamless functioning of computing systems.





Process	P1	P2	P3	P4	P5	P6	P7
Burst Time	3	5	4	2	9	4	10
Priority	3	6	3	5	7	4	10
Arrival Time	0	2	1	4	6	5	7

Figure 2.6 An illustration of the priority scheduling algorithm
Source: [8]

The choice of which process gets to be executed next is usually decided by a CPU scheduling algorithm, of which there are many with their own benefits and downfalls. Some of the common scheduling algorithms include [3]:

1. *First Come First Served (FCFS):* All processes are treated equally, and the earliest-arriving process is always next to be executed.
2. *Shortest Job First (SJF):* The next executed process is the one with the shortest execution time, therefore minimizing average wait times and maximizing processes finished per second.
3. *Priority Scheduling:* Each process is given a priority value, and the lower-priority processes get pushed to the front of the queue.
4. *Round Robin (RR):* A pre-emptive algorithm where each process gets executed for a certain amount of time, and if it is not finished yet after that the duration has passed, it will be dequeued and then pushed back to the tail of the queue to be finished later.

These scheduling algorithms can be classified into *non-preemptive* and *preemptive* algorithms. Non-preemptive algorithms are ones that require an executing process to be fully finished before moving on to the next process, while preemptive algorithms can partially execute a process and move to the next process, finishing the previous one at a later time.

The round robin algorithm is an example of a preemptive algorithm, while the rest that has been mentioned fall under the non-preemptive category.

III. Methodology

A. Limitations

With the theoretical frameworks of a binary heap and CPU scheduling established, this paper will now attempt to simulate CPU scheduling in Python. Simulating the inner workings of a CPU and the operating system is no easy task, therefore the method of simulation going forward will follow these few assumptions:

1. The simulated system is a *uniprocessor* system, therefore all processes will be queued and executed by the single processor.
2. The simulation will be *non-dynamic*, which means it cannot simulate processes queueing in and the process being executed simultaneously. Instead, the simulation will be divided into two phases, the *queueing in phase*, and the *execution* phase.
3. The simulation will only simulate *non-preemptive algorithms*, therefore each process will have to be fully completed before the CPU executes the next process in queue.

With these assumptions applied, the result of this paper might not reflect the results of real CPU scheduling, but it would still bring into light the benefits of the heap-based data structure and an overview of its effects on optimizing the scheduling process.

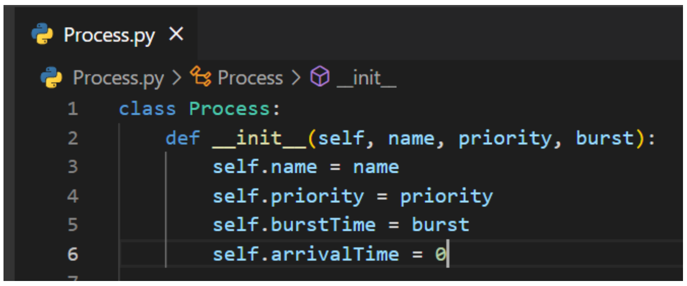
B. Tools

The tools that will be used for this simulation include:

1. Python 3.10
2. The *heapq* built-in library in Python for heap queue operations
3. *Jupyter Notebook, pandas, and matplotlib* for data analysis and visualization

C. The Process Class

The main function of the proposed program is to simulate the sorting of processes in the scheduling queue. Therefore, a *Process class* needs to be defined first. This process class will serve as the main element type that will be inserted and sorted inside the queue. Figure 3.1 shows the structure of the Process class that will be used in the program.



```python
class Process:
    def __init__(self, name, priority, burst):
        self.name = name
        self.priority = priority
        self.burstTime = burst
        self.arrivalTime = 0
```

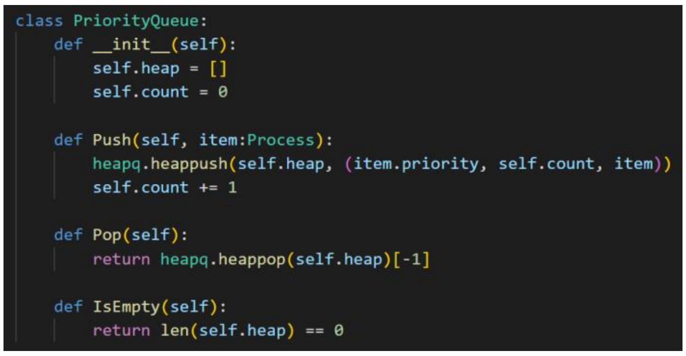
Figure 3.1 The Process class that represents the processes to be executed by the CPU
Source: Personal documentation

The Process class, upon initialization, will be given a *process name* (for identification purposes only), a *priority value* (lower priority processes will be executed first by the CPU, and a *burst* *time* (the time it takes the CPU to execute the process). An *arrival time* will also be automatically initialized to 0, but will later be replaced with the time the process arrives in the scheduling queue.

D. The Priority Queue

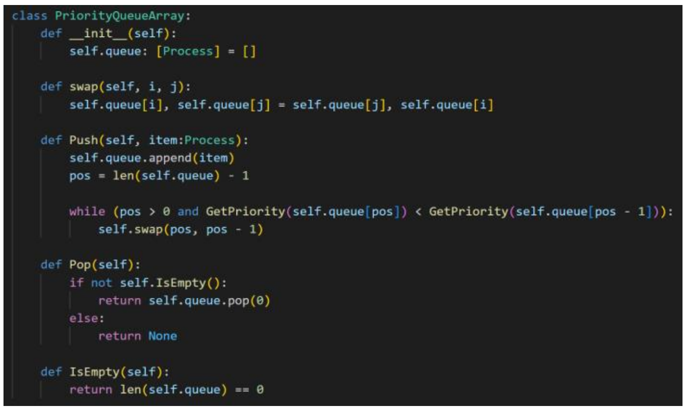
Aside from needing a Process class, the program obviously also needs a *Priority Queue class* to store and sort the processes for execution. In this simulation, we will be implementing both the *heap-based priority queue* and the *array-based priority queue*. This is such that we can compare the efficiencies of both implementations.

For the implementation of the *heap-based priority queue*, we will be utilizing the built-in *heapq* library with pre-made binary heap operations. Every implementation of a min-heap is essentially the same no matter the context, and since in this implementation we are more concerned with the benefits of a min-heap implementation in CPU scheduling instead of the min-heap itself, a manually-made heap data structure is not required.



```python
class PriorityQueue:
    def __init__(self):
        self.heap = []
        self.count = 0

    def Push(self, item:Process):
        heapq.heappush(self.heap, (item.priority, self.count, item))
        self.count += 1

    def Pop(self):
        return heapq.heappop(self.heap)[-1]

    def IsEmpty(self):
        return len(self.heap) == 0
```

(a)



```python
class PriorityQueueArray:
    def __init__(self):
        self.queue: [Process] = []

    def swap(self, i, j):
        self.queue[i], self.queue[j] = self.queue[j], self.queue[i]

    def Push(self, item:Process):
        self.queue.append(item)
        pos = len(self.queue) - 1

        while (pos > 0 and GetPriority(self.queue[pos]) < GetPriority(self.queue[pos - 1])):
            self.swap(pos, pos - 1)

    def Pop(self):
        if not self.IsEmpty():
            return self.queue.pop(0)
        else:
            return None

    def IsEmpty(self):
        return len(self.queue) == 0
```

(b)

Figure 3.2 (a) The Heap-based Priority Queue, (b) The Array-Based Priority Queue
Source: Personal documentation

Figure 3.2 shows the source code for both priority queue implementations. These implementations are functionally the same, behaving as a data structure where deletion is always from the front of the queue, and insertion starts from the back and then shifted to its appropriate location based on priority.

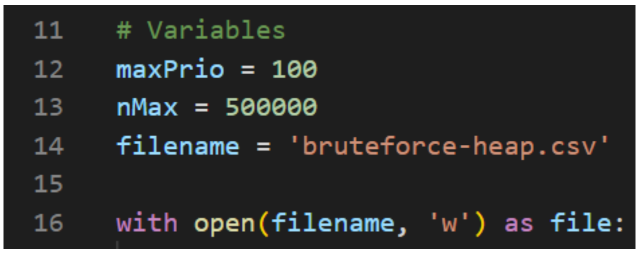
The main difference between both structures are the algorithms used to execute said operations. In the heap-based implementation, inserting to the correct position is always in *O(log n)* time, since the maximum amount of swaps required is the height of the binary tree. The array-based implementation, in contrast, might have to swap with every element in the queue in the worst-case scenario, therefore having *O(n)* insertion time.

E. The CPU Scheduling Simulator

With the needed data types and data structures already implemented, we are now ready to implement the CPU scheduling program. A brief overview of the flow of the program is as follows:

1. The inner program will simulate the CPU Priority Scheduling Algorithm for both the heap-based and array-based priority queue, for a finite N amount of processes.
2. The outer program receives a number N_{min}, N_{max}, and K, and for each value $N_i + K$ in the range [N_{min}, N_{max}] (inclusive), the outer program will run the inner program with $N = N_i$ amount of processes.
3. For each iteration of the inner program, the result is saved into a .csv file.

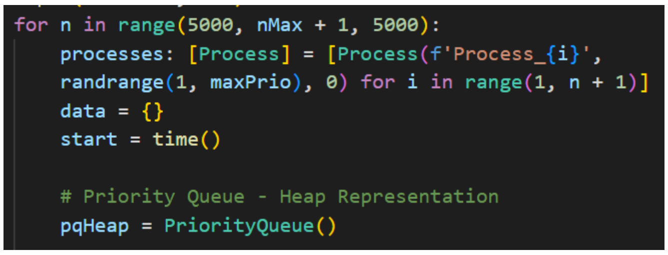
Further details of the program flow as well as the source code of the program is as follows:



```python
# Variables
maxPrio = 100
nMax = 500000
filename = 'bruteforce-heap.csv'

with open(filename, 'w') as file:
```

Figure 3.3 Program initialization
Source: Personal documentation

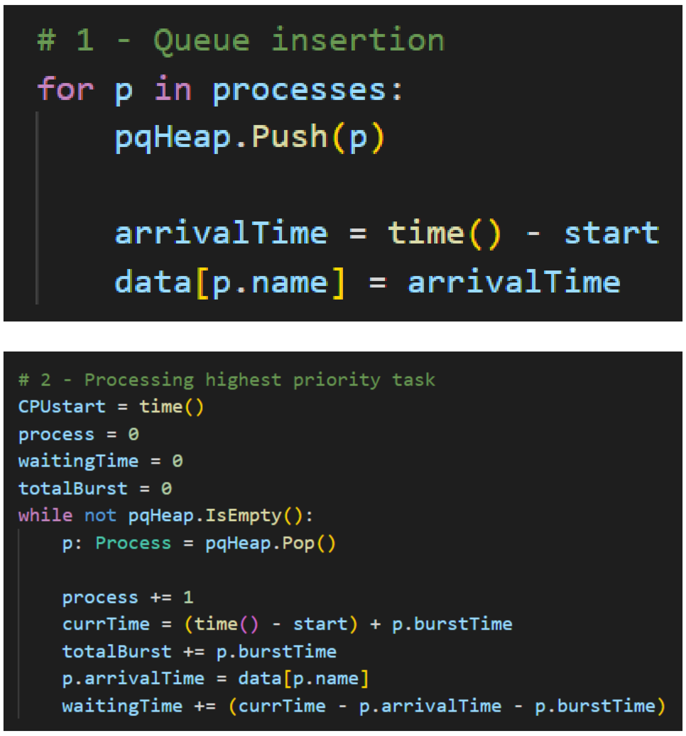
1. The program will first set a *max priority value*, N_{max}, and the filename to store the results. The file is then opened, and every line of code after this is executed inside the *with* block.



```python
for n in range(5000, nMax + 1, 5000):
    processes: [Process] = [Process(f'Process_{i}',
    randrange(1, maxPrio), 0) for i in range(1, n + 1)]
    data = {}
    start = time()

    # Priority Queue - Heap Representation
    pqHeap = PriorityQueue()
```

Figure 3.4 Starting the loop
Source: Personal documentation

2. The outer program starts a for loop for each $N_i + K$ in the range [N_{min}, N_{max}]. In the example in Figure 3.4, N_{min} = *5000*, and K = *5000.*
3. The inner program starts by initializing a list of N processes, and a *data* dictionary that will be used to store time values such as arrival time.
4. The start time of the iteration is stored in the *start* variable, and the priority queue is initialized.

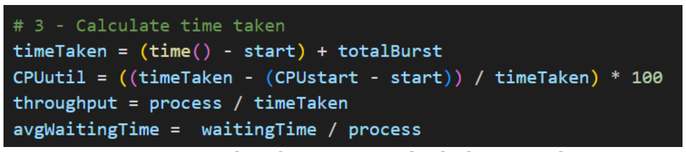


```
# 1 - Queue insertion
for p in processes:
    pqHeap.Push(p)

    arrivalTime = time() - start
    data[p.name] = arrivalTime
```

```
# 2 - Processing highest priority task
CPUstart = time()
process = 0
waitingTime = 0
totalBurst = 0
while not pqHeap.IsEmpty():
    p: Process = pqHeap.Pop()

    process += 1
    currTime = (time() - start) + p.burstTime
    totalBurst += p.burstTime
    p.arrivalTime = data[p.name]
    waitingTime += (currTime - p.arrivalTime - p.burstTime)
```

Figure 3.5 The process of filling the queue, and then emptying the queue
Source: Personal documentation

5. Each process in the *processes* list will now be inserted to the priority queue. This is the main section that is affected by the choice of the priority queue data structure.
6. After every process has been inserted to the queue, the CPU will "execute" each process, starting with the highest priority process, until the queue is empty. In the process, it will keep track of the total *burst time* and *waiting time*, for calculations later.



```
# 3 - Calculate time taken
timeTaken = (time() - start) + totalBurst
CPUutil = ((timeTaken - (CPUstart - start)) / timeTaken) * 100
throughput = process / timeTaken
avgWaitingTime =  waitingTime / process
```

Figure 3.6 Ending the iteration and calculating results
Source: Personal documentation

7. After steps 5 and 6 are finished, the simulation for the specific iteration is done, the statistics of CPU performance is calculated, and the results are saved to the .csv file.
8. The program then loops back to step 4, starting another iteration.

IV. Results and Analysis

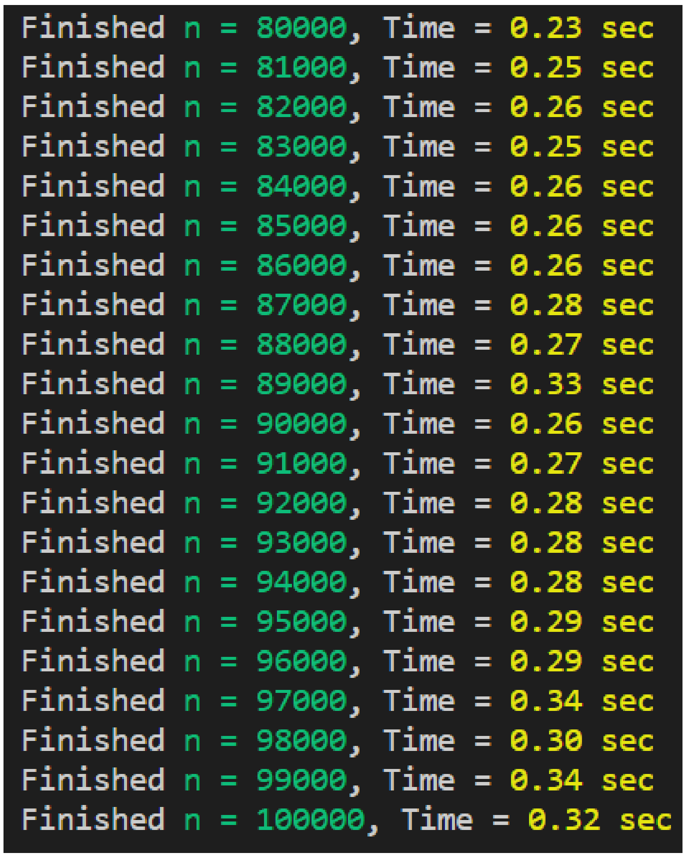
We will begin the experimentation process by defining the case studies that will be experimented and analyzed in this paper:

1. *Brute-forcing N Values*
2. *CPU Scheduling Performance*

In the first case study, we are only concerned with the time taken for the simulator to queue and execute *N* processes, therefore CPU-specific values such as burst time, throughput, etc. will be ignored. In the other case studies, however, we will be further analyzing how the algorithm effects these CPU-specific values.

A. Brute-forcing N Values

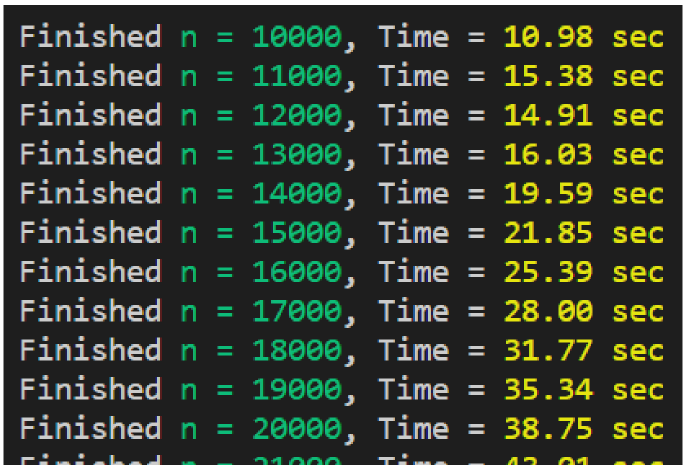
In this case study, we will brute force N values as high as possible and analyze the times taken for both priority queue implementations. Because we are only concerned with the speed of the simulation, the only variable we will be analyzing is the *total time taken to queue and execute N processes.*



```
Finished n = 80000, Time = 0.23 sec
Finished n = 81000, Time = 0.25 sec
Finished n = 82000, Time = 0.26 sec
Finished n = 83000, Time = 0.25 sec
Finished n = 84000, Time = 0.26 sec
Finished n = 85000, Time = 0.26 sec
Finished n = 86000, Time = 0.26 sec
Finished n = 87000, Time = 0.28 sec
Finished n = 88000, Time = 0.27 sec
Finished n = 89000, Time = 0.33 sec
Finished n = 90000, Time = 0.26 sec
Finished n = 91000, Time = 0.27 sec
Finished n = 92000, Time = 0.28 sec
Finished n = 93000, Time = 0.28 sec
Finished n = 94000, Time = 0.28 sec
Finished n = 95000, Time = 0.29 sec
Finished n = 96000, Time = 0.29 sec
Finished n = 97000, Time = 0.34 sec
Finished n = 98000, Time = 0.30 sec
Finished n = 99000, Time = 0.34 sec
Finished n = 100000, Time = 0.32 sec
```

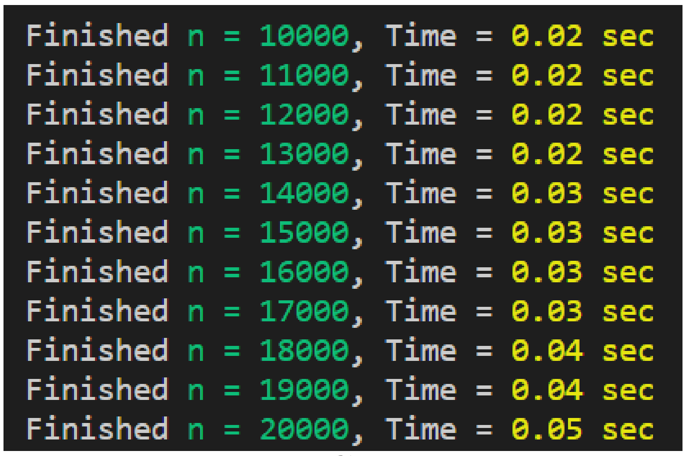
Figure 4.1 Terminal output of program while running
Source: Personal documentation

Figure 4.1 shows the output of the program to the terminal during execution. After one iteration is finished, the *N* value of the iteration and the total time taken for the iteration (in seconds) is displayed.



```
Finished n = 10000, Time = 10.98 sec
Finished n = 11000, Time = 15.38 sec
Finished n = 12000, Time = 14.91 sec
Finished n = 13000, Time = 16.03 sec
Finished n = 14000, Time = 19.59 sec
Finished n = 15000, Time = 21.85 sec
Finished n = 16000, Time = 25.39 sec
Finished n = 17000, Time = 28.00 sec
Finished n = 18000, Time = 31.77 sec
Finished n = 19000, Time = 35.34 sec
Finished n = 20000, Time = 38.75 sec
```

(a)



```
Finished n = 10000, Time = 0.02 sec
Finished n = 11000, Time = 0.02 sec
Finished n = 12000, Time = 0.02 sec
Finished n = 13000, Time = 0.02 sec
Finished n = 14000, Time = 0.03 sec
Finished n = 15000, Time = 0.03 sec
Finished n = 16000, Time = 0.03 sec
Finished n = 17000, Time = 0.03 sec
Finished n = 18000, Time = 0.04 sec
Finished n = 19000, Time = 0.04 sec
Finished n = 20000, Time = 0.05 sec
```

(b)

Figure 4.2 Terminal output results for higher N values, (a) Results for the array-based priority queue implementation, (b) Results for the heap-based priority queue implementation

Figure 4.2 shows the simulation results for higher N values. From first glance, it is glaringly obvious how high the array-based priority queue spikes in execution time for higher N values. Indeed, by plotting the execution time for both implementations in a line graph, we can see the astronomical rise of the array-based simulation.

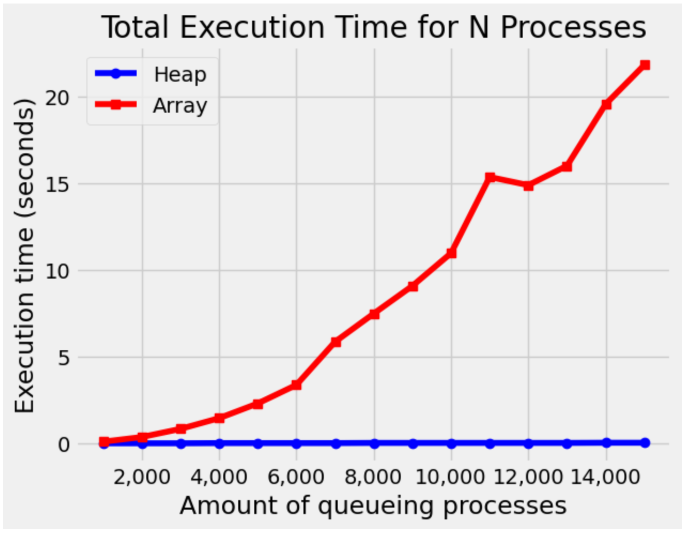


Figure 4.3 The rise of execution time for the array-based simulation Source: Personal documentation

Figure 4.3 shows the graph of total execution time in respect to N processes for both simulations. How to interpret this graph is: If the CPU has to queue and execute around 12,000 processes, the array-based simulation would finish executing all processes in around 15 seconds, while the heap-based simulation would finish in near 0 seconds.

A first thought might be that, while there's an obvious spike in execution time for the array-based simulation, the difference seems negligible, only differing about 20 seconds However, the graph in Figure 4.3 is not the full picture, or more precisely, it's only 37.5% of the full picture.

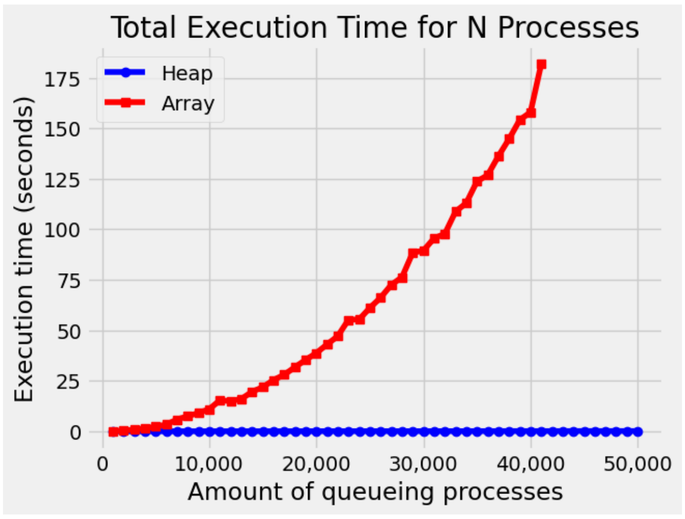


Figure 4.4 The full plotted graph of the experiment Source: Personal documentation

Figure 4.4 shows the plotting data for the entire result of both simulations. As you can see, the array graph spikes so high up in seconds resulting in the y-axis being so stretched out in scale, thus causing the heap graph to look flat in comparison.

The author has chosen to show the incomplete graph first to showcase that both graphs are indeed increasing, but for much bigger N sizes, the array-based priority queue spikes much quicker in execution time than the heap-based queue.

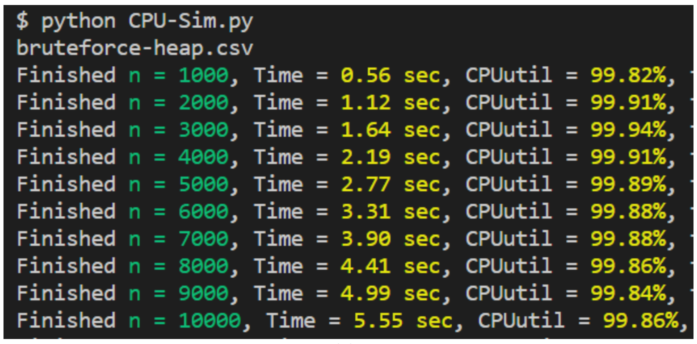
B. CPU Scheduling Performance

In this section, we will be analyzing how a change from a linear priority queue to a binary heap priority queue affects CPU scheduling performance. There are many criteria to dictate the quality of a CPU scheduler, but the ones of interest to us in this simulation are as follows:

1. *CPU Utilization:* The percentage of time where the CPU is executing processes. A CPU is said to have 100% utilization if it is executing a process in every single moment.
2. *Throughput:* The amount of processes being finished every second.
3. *Average Waiting Time:* The average time a process has to wait in queue. That is, for each process, the waiting time is the duration from which it enters the queue, to when it is finally executed by the CPU.

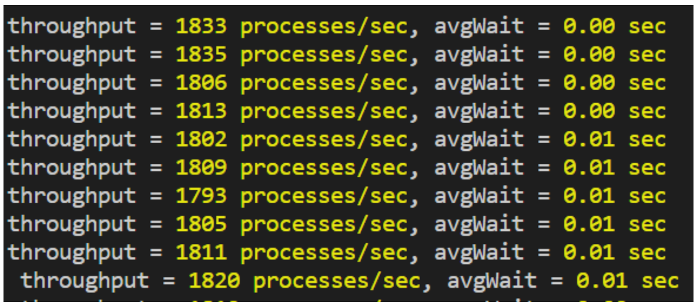
In the previous section we have ignored these criteria as well as other units such as burst rate, which before we have initialized as 0 to make computation easier. However, we will be introducing burst times to each process in this section, therefore a process will not be finished in an instant, it will take exactly *B* seconds to finish once it has been started executing by the CPU, where *B* is the burst time of that process

With burst times being introduced, computation times will increase, since each process are not finished instantly. Therefore, for this experiment, we have decreased N_{min} to 1000, N_{max} to 30000, and *K* to 1000.



```
$ python CPU-Sim.py
bruteforce-heap.csv
Finished n = 1000, Time = 0.56 sec, CPUutil = 99.82%,
Finished n = 2000, Time = 1.12 sec, CPUutil = 99.91%,
Finished n = 3000, Time = 1.64 sec, CPUutil = 99.94%,
Finished n = 4000, Time = 2.19 sec, CPUutil = 99.91%,
Finished n = 5000, Time = 2.77 sec, CPUutil = 99.89%,
Finished n = 6000, Time = 3.31 sec, CPUutil = 99.88%,
Finished n = 7000, Time = 3.90 sec, CPUutil = 99.88%,
Finished n = 8000, Time = 4.41 sec, CPUutil = 99.86%,
Finished n = 9000, Time = 4.99 sec, CPUutil = 99.84%,
Finished n = 10000, Time = 5.55 sec, CPUutil = 99.86%,
```

(a)



```
throughput = 1833 processes/sec, avgWait = 0.00 sec
throughput = 1835 processes/sec, avgWait = 0.00 sec
throughput = 1806 processes/sec, avgWait = 0.00 sec
throughput = 1813 processes/sec, avgWait = 0.00 sec
throughput = 1802 processes/sec, avgWait = 0.01 sec
throughput = 1809 processes/sec, avgWait = 0.01 sec
throughput = 1793 processes/sec, avgWait = 0.01 sec
throughput = 1805 processes/sec, avgWait = 0.01 sec
throughput = 1811 processes/sec, avgWait = 0.01 sec
 throughput = 1820 processes/sec, avgWait = 0.01 sec
```

(b)

Figure 4.5 The new format of terminal output, with added values
Source: Personal documentation

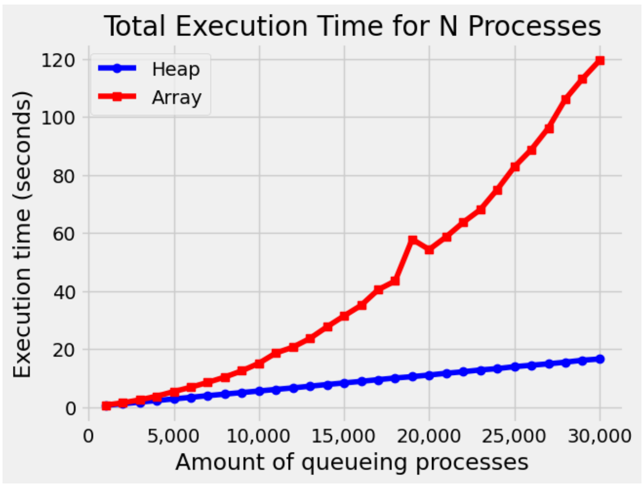


Figure 4.6 The new execution time from adding burst times to processes
Source: Personal documentation

With burst times now being implemented, total execution time have increased a significant amount, as seen on Figure 4.6. With this newly modified simulator, though, we are able to extract more information about CPU performance.

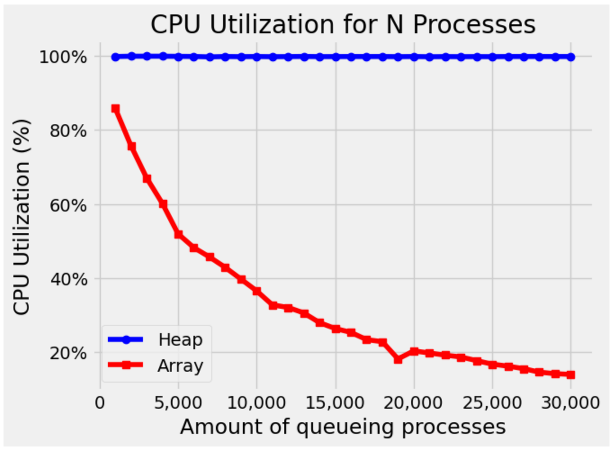


Figure 4.7 CPU Utilization Graph
Source: Personal documentation

CPU utilization refers to the ratio of time where the CPU is executing processes to the total recorded time. If a CPU has near 100% utilization, it means that it is almost never idle and is executing processes at any given moment.

In this simulation, all processes need to be already in the queue before the CPU starts executing them. In the case of the heap-based simulation, it finishes the queue insertion part almost instantly, and therefore has near 100% CPU utilization.

The array-based simulation, however, is already at 80% utilization even at N = 1000 processes queueing. This number rapidly declines until going under 20% at around 20,000 processes.

The rapid decline in utilization is because the array-based simulation spends a very long time inserting processes into the queue. At N = 20,000 processes, the CPU is almost 80% idle at any given moment, simply waiting for the queue to be finished sorting.

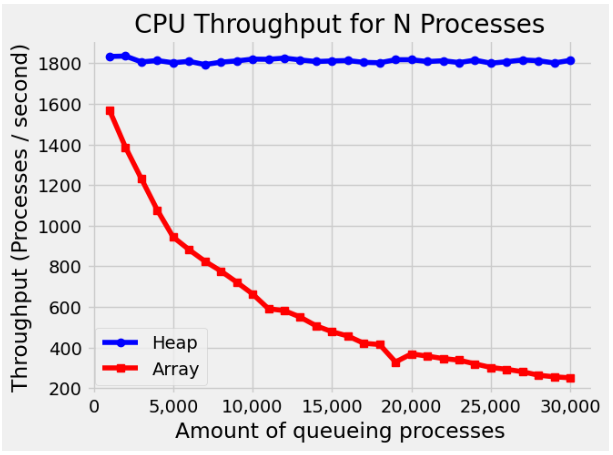


Figure 4.8 CPU Throughput Graph
Source: Personal documentation

CPU throughput refers to the amount of processes being finished every second (processes / sec). If a CPU has high throughput, it means that it is has rapid speed in finishing processes.

Using our simulation, there shouldn't be much difference between execution speed between both array-queue and heap-queue, since deletion in the queue is very fast for both.

What makes the array simulation steadily declining, though, is simply attributed to the extended time inserting into the queue. When processes are being inserted to the queue, the CPU is idle and therefore not finishing any tasks. This is why the graph for throughput follow a similar decline as with the CPU utilization graph.

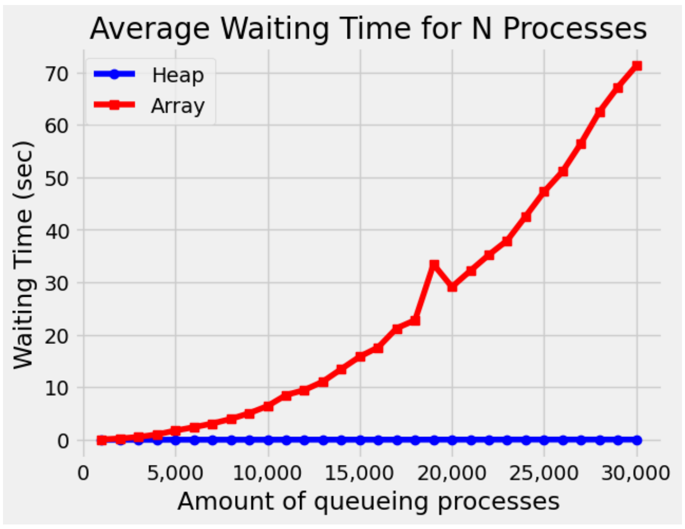


Figure 4.9 Average Waiting Time Graph
Source: Personal documentation

For the last scheduling criteria, we have average waiting time. The waiting time of a process refers to the duration it has to wait in the queue before being executed. If a scheduler has low average waiting times, it means that a process usually gets executed very quickly after entering the queue.

This graph almost looks like an inverse of the two graphs previously discussed, and the reasoning for this is pretty much the same. Because the insertion phase takes very long in the array-based simulation, the execution phase starts even longer as well, thus for the processes in the queue, a lot of time is spent simply waiting for the insertion phase to finish before actual execution occurs.

V. Conclusion

From the results of the CPU scheduling simulator, it is evidently clear how using binary trees instead of linear arrays as priority queues significantly improve the efficiency of process scheduling. This case study shows a classic example of the vast difference a simple data structure change can make to a certain process.

This study has shown that the concept of the mathematical tree in discrete mathematics, while simple, can bring huge benefits to real life problems. A data structure for a process scheduling program is simply one of many. Therefore, the author kindly invites the reader to also find interesting and innovative applications of discrete mathematics concepts in other problems, and write a paper of analysis discussing it, inviting others to do the same.

As for the technicalities of this experiment, the author recognizes that it is far from perfect, and has large rooms for improvement. An example of imperfection in this implementation is the huge distinction of the simulated CPU to real-life CPUs. If given opportunities in the future, the author would like to improve this experiment to better reflect and simulate the real-life problem.

VI. Appendix

The complete source code for this project can be found at this repository: https://github.com/trimonuter/CPU-Sim

VII. Acknowledgment

The author would like to deeply thank Mr. Dr. Ir. Rinaldi Munir, M.T., and Mr. Monterico Adrian, S.T., M.T. as the author’s lecturers of Discrete Mathematics, and by extension, the entire Discrete Mathematics staff, consisting of lecturers and assistants, for giving the author a chance to not only deepen their knowledge of the field, but to conduct this study as well. Lastly, but certainly not least, the author would like to thank their friends and families, for always giving them support and always being present while going through every hardship experienced in the process of conducting their study as well as writing this academic paper.

References

[1] Rosen, Kenneth. (2012). *Discrete Mathematics and its Applications, Seventh Edition*. McGraw-Hill International.
[2] *Trees in Data Structure*. Programiz, https://www.programiz.com/dsa/trees. Accessed 8 December 2023, 10:22 PM.
[3] Stallings, W. (2013). *Operating Systems: Internals and Design Principles, Seventh Edition*. Pearson Education Limited.
[4] *Priority Queue*. Programiz, https://www.programiz.com/dsa/priority-queue. Accessed 9 December 2023, 6:20 AM.
[5] *Heap Data Structure*. GeeksForGeeks, https://www.geeksforgeeks.org/heap-data-structure. Accessed 9 December 2023, 6:20 AM.
[6] *Binary Tree*. Programiz, https://www.programiz.com/dsa/binary-tree. Accessed 9 December 2023, 6:58 AM.
[7] *Wengrow, J.* (2017). *A Common-Sense Guide to Data Structures and Algorithms: Level Up Your Core Programming Skills*.
[8] *Priority Scheduling Algorithm.* Prepinsta, https://prepinsta.com/operating-systems/priority-scheduling-algorithm/. Accessed 10 December 2023, 6:46 PM.

STATEMENT

I hereby declare that this paper I have written is my own work, not a translation or adaptation of someone else's paper, and is not plagiarized.

Bandung, December 10th, 2023

[illegible]

Muhammad Rasheed Qais Tandjung
13522158