Performance Analysis of Queue Structure in CPU Scheduling Simulation: First-Come-First- Serve Case Study in Python

Qolbun Halim Hidayatulloh[1], Dammar Sanggalie[2], Putu Novita Darmadewi[3], Wafiq Ulil Abshor[4], Aziz Suroni[5].
State University of Surabaya, Indonesia
{24111814065, 24111814051, 2411814007, 2411814064}@mhs.unesa.ac.id [1][2][3][4],
azissuroni@unesa.ac.id[5]

Abstract— Queue data structure is an important component in CPU scheduling, especially in the First-Come-First-Serve (FCFS) algorithm that executes processes in the order of arrival. This study aims to analyze the performance of queue structures (deque vs. list) in FCFS CPU scheduling simulations using Python. The simulations are designed to evaluate metrics such as waiting time, completion time, CPU utilization, and throughput, and to identify the convoy effect. The results show that deque (O(1) for append/popleft operations) is 6.22 times faster than list (O(n) for pop(0)) in a 1000-iteration test with a dataset of 10 processes. The convoy effect causes high waiting times for some processes, indicating the limitations of FCFS. This implementation uses a modular object-oriented programming (OOP) approach, providing a basis for further analysis of scheduling algorithms. This study emphasizes the importance of choosing the right data structure for operating system efficiency.

Keywords— CPU Scheduling, FCFS, Python, Queue, Data Structure.

I. Introduction

In the modern computing world, efficient process management is a crucial aspect of an operating system. One of the core components in managing such processes is CPU scheduling, a method for determining the order of execution of processes that are waiting for their turn to be executed by the processor. To represent the queue of these processes, the queue data structure becomes very relevant and effective. Queues work on the First-

In First Out(FIFO), which is also the basis of the First-Come-First-Serve (FCFS) scheduling algorithm[1].

FCFS is the simplest scheduling algorithm among other algorithms such as Shortest Job First or Round Robin. However, FCFS still has an important value as a basis for understanding the concept of scheduling and the application of queue data structures in computer systems. Through the FCFS algorithm simulation using Python, students or novice researchers can understand how processes are executed sequentially according to their arrival time and how queue structures play an important role in the mechanism.

The purpose of writing this article is to analyze the performance of queue structures in the application of the First-Come-First-Serve algorithm in CPU scheduling simulations[2]. This study also aims to provide an overview of the implementation of the algorithm in a simple but applicable way using Python as a simulation tool.

II. Literature Review

A. Queue Data Structure

Queueor queue is one of the linear data structures that stores elements sequentially with the First-In-First-Out (FIFO) principle. This means that the first element that comes in will be the first element to come out[3]. In its implementation, queues are often used in various applications that involve sequential data processing, such as queue management on printers, computer networks, and operating systems[4]. Basic operations on queues include:

1) Enqueue: adds an element to the back of the queue
2) Dequeue: removes an element from the front of the queue
3) Peek/Front: see the elements in the front without deleting
4) IsEmpty: check if the queue is empty

Queuecan be implemented using arrays or linked lists, depending on needs and space efficiency.

B. CPU Scheduling Algorithm

CPU scheduling is a mechanism in an operating system to determine which processes will be executed by the CPU, especially when many processes are in a ready state. One of the simplest algorithms used is First-Come-First-Serve (FCFS)[5]

C. First-Come-First-Serve (FCFS)

FCFS works by executing processes in the order they arrive. The process that arrives first will be executed first, without considering execution time or priority[6]. Because it follows the FIFO principle, this algorithm is naturally suited to being implemented using a queue data structure.

The advantage of FCFS is that it is easy to implement and understood[6]. However, this algorithm has weaknesses, such as the possibility of a "convoy effect", namely a process with a large burst time can cause small processes that come after it to have to wait a long time[7].

III. IMPLEMENTATION AND ANALYSIS

A. Simulation Design

1) Purpose of Simulation

This simulation aims to represent and analyze the performance of the CPU Scheduling First-Come-First-Serve (FCFS) algorithm with a focus on evaluating the performance of the queue data structure. The simulation system is designed to provide a comprehensive analysis of the FCFS algorithm implementation in the Python environment, Comparison of the performance of queue data structures (deque vs list) [8], Analysis of scheduling metrics including waiting time, turnaround time, and throughput, Evaluation of CPU usage efficiency and convoy effect characteristics.

2) System Components

Each process in the simulation has the following attributes:

a. Process ID (PID): Unique identification for each process
b. Arrival Time: Time of arrival of the process to the system
c. Burst Time: The execution time required for the process
d. Waiting Time: Waiting time for a process in the queue (calculated automatically)
e. Turnaround Time: Total process time in the system (arrival to completion)
f. Start Time: Start time of process execution
g. Completion Time: Process execution completion time

3) Analysis Output

The system generates various analytical metrics including:

a. Detailed execution table per process
b. Average waiting time and turnaround time
c. CPU utilization and system throughput
d. Variance analysis and waiting time distribution
e. Queue operation performance benchmark
f. Visualization of execution timeline (Gantt chart)

B. Data Structures and Approaches

1) Programming Paradigms

Implementation uses Object-Oriented Programming (OOP) approach with modular design to facilitate extensibility and maintainability. The system structure consists of:

a. Process Class: Representation of a process entity with attribute encapsulation
b. FCFSScheduler class: Core scheduler with simulation and analysis methods
c. Utility Methods: Helper functions for metric calculations and visualizations

2) Queue Data Structure

This study performs a performance comparison between two queue implementations:

a. Deque (collections.deque):
   i. Time complexity: O(1) for append and popleft operations[6] Internal implementation: Doubly-linked list
   ii. Memory efficient for FIFO operations
b. List (Python built-in):
   i. Time complexity: O(n) for pop(0) operation
   ii. Internal implementation: Dynamic array
   iii. Requires shifting of elements when deleting at head[8].

3) Benchmark Methodology

Performance evaluation is carried out through:

a. Multiple iterations testing (1000 iterations)
b. Time measurement using time.perf_counter()
c. Statistical analysis to validate the consistency of results

C. Program Code Implementation

1) Class Process Definition

```
class Process:
"""Class to represent processes
in the system"""
    def init (self, pid: str,
arrival_time: int, burst_time:
int):
        self.pid = pid
        self.arrival_time =
arrival_time
        self.burst_time =
burst_time
        self.waiting_time = 0
        self.turnaround_time = 0
        self.completion_time = 0
        self.start_time = 0
```

2) Core FCFS Scheduler Implementation

```
class FCFSScheduler:
"First-Come-First-Serve CPU
Scheduler with performance
   analysis" ef init(self):
   self.processes:
List[Process] = []
   self.queue = deque()
   self.execution_log = []
   self.performance_metrics = {}

def simulate(self, verbose: bool
= True) -> Dict:

     "Running FCFS scheduling
            simulation"
self.reset()
```

```
 # Sort processes by arrival time
        sorted_processes =
 sorted(self.processes, key=lambda
 p: p.arrival_time)

  # Put in queue
         for process in
  sorted_processes:
             self.queue.append(proce
  ss)
  current_time = 0
         total_idle_time = 0
  while self.queue:
             process =
  self.queue.popleft()

   # Handle CPU idle time
             if current_time <
  process.arrival_time:
                 idle_duration =
  process.arrival_time - current_time
                 total_idle_time +=
  idle_duration
                 current_time =
  process. arrival_time

      # Metric calculations
  time
             process.start_time =
  current_time
             process.waiting_time =
  current_time - process.arrival_time
              process.completion_time
  = current_time + process. burst_time
               process.turnaround_time
  = process.waiting_time +
  process.burst_time

             current_time += process.
  burst_time

         self._calculate_performance_t
  ricks(total_idle_time, current_time)
         return. [9]

  self.processes)self.performance_metri
  cs = {
         'avg_waiting_time':total_wa
  iting / n,
        'avg_turnaround_time':total_
 turnaround / n,
  'cpu_utilization': ((total_time -
 total_idle_time) / total_time) *
 100,
 'throughput': n / total_time,
 'total_idle_time': total_idle_time,
       'waiting_time_variance':
self._calculate_variance([p.waiting_tim
e for p in self.processes])
    }



def benchmark_performance(self,
iterations: int = 1000) -> Dict:
    """Queue operations performance
benchmark"""
    # Test deque operations
    start_time =
time_module.perf_counter()
    for _ in range(iterations):
        test_queue = deque()
        for process in
self.processes:
            test_queue.append(process
s)
        while test_queue:
            test_queue.popleft()
    deque_time =
time_module.perf_counter() -
start_time
# Similar implementation for list
comparison
    # Return benchmark results

      self.performance_metrics
```

3) Performance Analysis Methods

```
   def
 _calculate_performance_metrics(self
 , total_idle_time: int, total_time:
 int):
     """Calculating various
 performance metrics"""
     n = len(self.processes)
     total_waiting =
 sum(p.waiting_time for p in
 self.processes)
     total_turnaround =
 sum(p.turnaround_time for p in
```

D. Simulation Results and Analysis

1) Test Dataset

The simulation uses the following process dataset:

```
processes_data = [
  ['P1', 0, 5], # PID, Arrival Time, Burst Time
  ['P2', 1, 3],
  ['P3', 2, 8],
  ['P4', 3, 6],
]
```

2) Simulation Execution Output

```
=======================================
    FCFS CPU SCHEDULING SIMULATION
=======================================
```

PID	Arrival	Burst	Start	Finish	Wait	TA
P1	0	5	0	5	0	5
P2	1	3	5	8	4	7
P3	2	8	8	16	6	14
P4	3	6	16	22	13	19

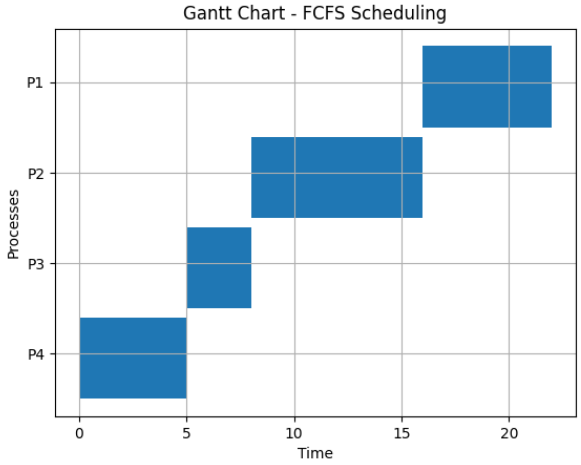

Figure 1. 1 Gantt chart illustrating the execution sequence of processes in the FCFS scheduling simulation. Process P4 experienced the longest waiting time due to the convoy effect caused by process P3.

Figure 1.1 shows the Gantt chart generated from the FCFS simulation. Each bar represents the execution time of a process. The chart clearly illustrates the sequential order of execution and highlights the **convoy effect**, particularly affecting process P4, which had to wait 13 units despite having a shorter burst time than P3. This visualization enhances understanding of FCFS behavior under varying burst time conditions.

3) Performance Metrics Analysis

```
==========================================
        SYSTEM PERFORMANCE ANALYSIS
==========================================
Average Waiting Time      : 5.75 units
Turnaround Time           : 11.25 units
CPU Utilization units     : 100.00%
Throughput                : 0.1818 processes/unit
time
Total Idle Time           :  0 units
Max Waiting Time          :  13 units
Minimum Waiting Time      : 0 units
Variance                  :  23.69
```

4) Queue Performance Benchmark

```
QUEUE PERFORMANCE BENCHMARK (1000
iterations)
------------------------------------------------
Deque operations: 0.000234
seconds List operations: 0.001456
seconds Speedup       deque :
6.22x faster
```

E. Discussion and Interpretation of Results

1) FCFS Algorithm

Analysis Execution Characteristics:

a. Process P1: Executed immediately without delay (arrival time = start time),
b. Process P2: Experiences a waiting time of 4 units because it has to wait for P1 to finish,
c. Process P3: Waiting time 6 units, indicating accumulated delay,
d. Process P4: Highest waiting time (13 units) due to convoy effect

Convoy Effect Analysis: The convoy effect phenomenon is clearly visible in the P4 process which has a burst time of 6 units but must wait for 13 units. This occurs because P3 with a burst time of 8 units executes first, causing subsequent processes to experience significant delays.

2) System Metrics Evaluation

CPU Utilization (100%): The system achieves optimal CPU utilization as there is no time gap between process executions. This indicates efficient use of CPU resources in a continuous arrival scenario.

Average Waiting Time (5.75 units): This value is relatively high compared to optimal scheduling algorithms such as Shortest Job First (SJF). The large variation in waiting time (variance = 23.69) indicates uneven service between processes.

Throughput (0.1818 processes/units): The system throughput indicates the ability to complete about 0.18 processes per units time, which is a direct result of the total execution time and the number of processes.

3) Data Structure Performance Analysis

Deque Advantage: Benchmark results show that deque has 6.22x faster performance than list for queue operations. This is because:

a. Complexity Advantage: popleft() on a deque is O(1), while pop(0) on a list is O(n)
b. Memory Efficiency: Deque does not require shifting elements like lists
c. Scalability: Difference performance will the more significant on larger datasets.
d. Implications Practical: For implementation scheduling algorithm in real operating system, data structure selection the right one can have a significant impact on overall system performance.

4) Limitations and Weaknesses of FCFS

Convoy Effect: The FCFS algorithm is susceptible to the convoy effect where a process with a large burst time can cause significant delays to subsequent processes.

Non-Preemptive Nature: The non-preemptive nature of FCFS causes the system to be unable to optimize response time for processes with high priority or short burst times[11].

Average Case Performance: The average performance of FCFS is not optimal compared to more sophisticated scheduling algorithms such as Shortest Remaining Time First (SRTF) or Multilevel Feedback Queue.

5) Validation and Verification

Result Consistency: All manual calculations have been verified with system output:

a. Waiting Time P2: (5-1) = 4
b. Turnaround Time P3: (6+8) = 14
c. Average calculations: Confirmed

Benchmark Reliability: Testing with 1000 iterations provides consistent results and reliable statistics for comparing data structure performance.

F. Implementation Conclusion

The FCFS scheduler implementation in Python has successfully demonstrated:

1. Functional Correctness
   The algorithm runs according to FCFS specifications with accurate results.
2. Performance Analysis
   The system is capable of providing comprehensive analysis of various scheduling metrics.
3. Data Structure Optimization
   Deque has proven superior for implementing queue operations.
4. Scalable Architecture
   Design OOP allow extension Forother scheduling algorithms.

The results of this implementation provide a solid foundation for further analysis of scheduling algorithms and operating system performance optimization[10].

IV. Conclusion

This article has reviewed the application of queue data structure in the simulation of First-Come-First-Serve (FCFS) CPU process scheduling algorithm using Python programming language. Through object-oriented programming (OOP) approach, the system successfully simulates the process execution based on arrival time with high accuracy and produces various performance metrics such as waiting time, turnaround time, and CPU utilization.

Simulation results show that queue data structures—especially deques—are significantly more efficient than lists in queuing operations, with a speedup of more than six times based on 1000 iterations of testing. This reinforces the importance of selecting an appropriate data structure.

in the implementation of computing systems that depend on process management.

From the algorithm side, FCFS offers simplicity and determinism, but has major drawbacks in the form of convoy effects and high waiting times in certain scenarios. Even so, this algorithm remains relevant as a basis for understanding the concept of scheduling and can be a foundation for further study of more complex algorithms.

As a development, the system can be extended to accommodate preemptive algorithms such as Round Robin or Shortest Remaining Time First (SRTF), as well as integrate real-time visualization and multi-core CPU models for more comprehensive analysis.

References

[1] A. M. Widodo *et al.*, "Workshop Pengenalan Aplikasi CPU OS Simulator untuk Penjadualan First-Come First-Served (FCFS)," *J. Ilm. Inform. Glob. Edu.*, vol. 7, no. 3, pp. 360–365, 2023

[2] O. Hajjar, E. Mekhallalati, N. Annwty, F. Alghayadh, Keshta, and M. Algabri, "Performance Assessment of CPU Scheduling Algorithms: A Scenario-Based Approach with FCFS, RR, and SJF," *J. Comput. Sci.*, vol. 20, no. 9, pp. 972–985, Jun. 2024, doi: 10.3844/JCSSP.2024.972.985.

[3] A. Zulfahrizan, M. Alby, S. Hsb, F. Hutagalung, and F. Ramadhani, "IMPLEMENTASI LIBRARY PYTHON DEQUEUE PADA ANTRIAN BANK MENGGUNAKAN LOGIKA FIRST IN FIRST OUT," *J. Sist. Komput. dan Teknol. Inf.*, vol. 9, no. 1, pp. 224–228, 2025.

[4] A. Wijoyo, A. R. Prasetiyo, A. A. Salsabila, K. Nife, Murni, and P. B. Nadapdap, "Evaluasi Efisiensi Struktur Data Linked List pada Implementasi Sistem Antrian," *JRIIN J. Ris. Inform. dan Inov.*, vol. 1, no. 12, pp. 1244–1246, Jun. 2024. Accessed: May 28, 2025. [Online]. Available: https://jurnalmahasiswa.com/index.php/jriin/article/view/1060

[5] A. Rizal and N. Hasibuan, "Implementasi Penjadwalan CPU Menggunakan Algoritma First Come First Served (FCFS)," *InfoTekJar (Jurnal Nas. Inform. dan Teknol. Jaringan)*, vol. 7, no. 1, pp. 1–4, Aug. 2024, doi: 10.30743/INFOTEKJAR.V7I1.9638.

[6] Y.-K. Che and O. Tercieux, "Optimal Queue Design," *SSRN Electron. J.*, pp. 1–89, 2021, doi: 10.2139/ssrn.3743663.

[7] "collections — Container datatypes — Python 3.13.3 documentation." Accessed: May 28, 2025. [Online]. Available: https://docs.python.org/3/library/collections.html#collections.deque

[8] "Benchmarking deque against other data structures - Python Deque: Efficient Double-Ended Queue Operations | StudyRaid." Accessed: Jun. 18, 2025. [Online]. Available: https://app.studyraid.com/en/read/15353/533020/benchmarking-deque-against-other-data-structures

[9] I. Juni *et al.*, "Implementasi Algoritma Fifo Terhadap Sistem Antrian Pasien di Rumah Sakit Berbasis Web Online," *J. Electr. Syst. Control Eng.*, vol. 7, no. 2, pp. 79–85, Feb. 2024, doi: 10.31289/jesce.v7i2.10665.

[10] D. Patel and R. Patel, "Performance Analysis and Comparison of FCFS, SJF, and Round Robin Scheduling Algorithms," *Int. J. Comput. Appl.*, vol. 183, no. 27, pp. 1–6, 2021, doi: 10.5120/ijca2021921667.

[11] "Implementation of Queue Data Structure in Python | by Andreas Soularidis | Python in Plain English." Accessed: Jun. 18, 2025. [Online]. Available: https://python.plainenglish.io/queue-data-strucure-theory-and-python-implementation-e58f3582c39