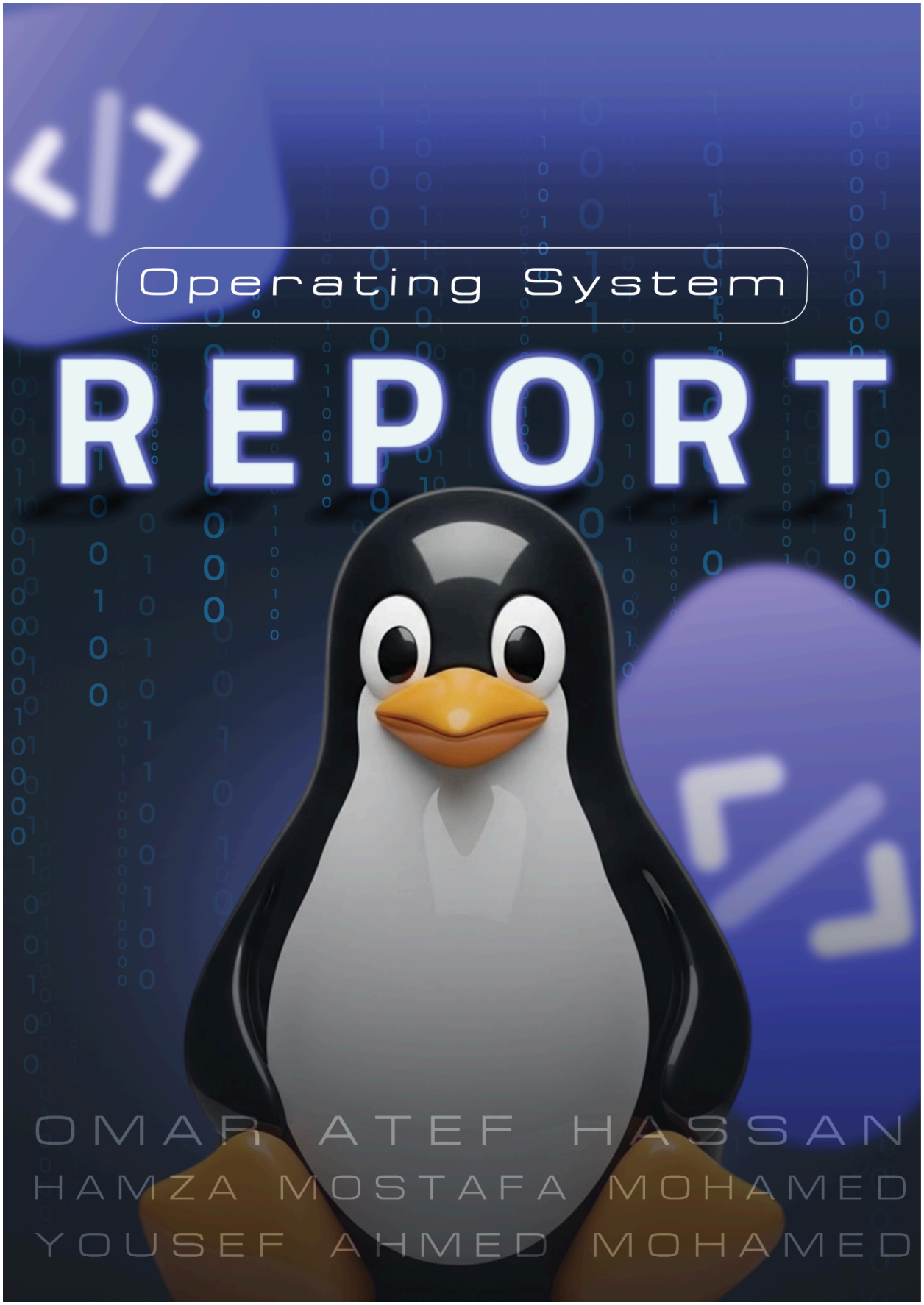
Operating System
REPORT
OMAR ATEF HASSAN
HAMZA MOSTAFA MOHAMED
YOUSEF AHMED MOHAMED


Team Members
عمـر عاطـف حسـن
حمـزه مصطفـي محمـد
يوسـف محمـد احمـد

Project Report

Non-Preemptive Shortest Job First (SJF) Scheduling :

1. Introduction

CPU scheduling is a core component of an Operating System, responsible for deciding which process runs at any given time. This project focuses on the implementation of the **Shortest Job First (SJF)** algorithm in its **Non-Preemptive** mode.

The primary objective of this code is to calculate key scheduling metrics—specifically Turnaround Time (TAT) and Waiting Time (WT)—to evaluate the efficiency of the schedule. SJF is chosen for its theoretical optimality in minimizing the average waiting time for a given set of processes.

2. Theoretical Background

2.1 Shortest Job First (SJF)

SJF is an algorithm that selects the process with the smallest execution time (Burst Time) to execute next.

2.3 Key Metrics

To evaluate the performance, the following standard formulas are used:

- Turnaround Time (TAT) = Completion Time (CT) - Arrival Time (AT)
- Waiting Time (WT) = Turnaround Time (TAT) - Burst Time (BT)

3. Algorithm Implementation Details

3.1 Data Structures

The solve method accepts a list of dictionaries (processes), where each dictionary represents a process containing:

- pid: Process ID
- at: Arrival Time
- bt: Burst Time

Internal arrays are initialized to store ct, tat, wt, and a boolean array completed to track the status of every process.

3.2 Logic Flow (The Algorithm)

1. **Time Simulation:** The variable current_time acts as the system clock, starting at 0.
2. **Process Selection:** Inside a while loop (which runs until all processes are complete), the code scans all processes.
3. **Criteria Check:** A process is a candidate for execution if:
    - It has arrived (at <= current_time).
    - It has not been completed yet (not completed[i]).
4. **Greedy Selection:** Among the candidates, the algorithm selects the process with the **minimum Burst Time**.
    - *Tie-Breaking:* If two processes have the same burst time, the code selects the one that arrived earlier (at[i] < at[idx]).
5. **Execution:**
    - If a valid process is found, current_time increases by that process's burst time. Metrics (ct, tat, wt) are calculated.
    - If *no* process has arrived yet (CPU idle), current_time is incremented by 1 to move time forward.

4 Output Generation

The function returns a dictionary containing:

1. **Data:** A detailed list including calculated CT, TAT, and WT for each process.
2. **Averages:** The Average Turnaround Time (avgTat) and Average Waiting Time (avgWt), rounded to 4 decimal places.

5. Complexity Analysis

5.1 Time Complexity

The algorithm uses a nested loop structure:

- The while loop runs N times (where N is the number of processes) because one process is completed per iteration.
- The inner for loop iterates through all N processes to find the minimum burst time.
- **Total Time Complexity :** O(N^2)

5.2 Space Complexity

The algorithm creates list arrays (at, bt, ct, tat, wt, completed) of size N.

- **Total Space Complexity:** O(N)

6. Limitations

While SJF is optimal for minimizing average waiting time, this implementation (and the algorithm in general) suffers from specific drawbacks:

1. **Starvation:** Long processes may wait indefinitely if shorter processes keep arriving.
2. **Prediction:** In real-world OS scenarios, the Burst Time is rarely known in advance; it must be estimated.

Preemptive Shortest Job First (SJF) Scheduling

1. Introduction

This project implements the **Preemptive Shortest Job First (SJF)** scheduling algorithm, often called **Shortest Remaining Time First (SRTF)**.

2. The Core Idea

The logic of this code is based on one golden rule: **Always work on the task closest to completion.**

How it works in Real Life:

Imagine you are at a printer printing a 100-page document.

1. Suddenly, a colleague runs in with a 1-page document.
2. **In Non-Preemptive:** You force them to wait until your 100 pages are done.
3. **In Preemptive (This Code):** You pause your printing at page 10, let them print their 1 page, and then resume your work.

3. Code Explanation

The Python code implements this using a Time Step approach. Here is the breakdown of the logic:

A. The Setup (rt vs bt)

In the code, we create a copy of the Burst Time (bt) called **Remaining Time (rt)**.

Python

```
rt = bt.copy()
```

- **Why?** Because in this algorithm, the time required changes every second as the CPU works. We need to track how much is *left*, not just how much it started with.

B. The Main Loop (The Clock)

The while loop acts as the system clock. Unlike simpler algorithms that jump forward in time,

This allows the system to re-evaluate the queue every single second to see if a better (shorter) process has arrived.

C. The Decision Logic

Inside the loop, the code looks at every available process to find the "Shortest":

1. **Is it here?** (at <= current_time)
2. **Is it unfinished?** (rt > 0)
3. **Is it the fastest?** (rt < min_rt)

If it finds a process with a smaller rt than the one currently running, it automatically switches (updates the shortest index) for the next second.

4. Test Example (Dry Run)

To prove the code works, let's trace it with a simple example.

Input:

- **P1:** Arrives at 0, Needs 5 seconds.
- **P2:** Arrives at 1, Needs 2 seconds.

Execution Trace:

1. **Time 0:** Only P1 is here. CPU starts P1.
    - P1 Remaining: 4.
2. **Time 1:** P2 arrives.
    - Compare: P1 needs 4s vs P2 needs 2s.
    - **Action:** CPU **preempts** (pauses) P1 and switches to P2.
3. **Time 2:** P2 runs. P2 Remaining: 1.
4. **Time 3:** P2 runs. P2 Remaining: 0. **P2 Done.**
5. **Time 3:** P2 is gone. CPU goes back to P1.

6. **Time 7:** P1 finishes remaining work. **P1 Done.**

5. Performance Analysis

The code calculates the standard metrics using these formulas:

- Turnaround Time (TAT): Total time from arrival to finish.
  TAT = CompletionTime - ArrivalTime
- Waiting Time (WT): Total time spent doing nothing.
  WT = TAT - BurstTime

Advantages of this Code:

1. **Optimal Average Waiting Time:** This algorithm is mathematically proven to have the lowest average waiting time of any scheduling algorithm.
2. **Responsiveness:** Short tasks are handled immediately, so the system feels faster to users.

Limitations:

1. **Overhead:** Because the code checks the queue every single second (time += 1), it uses more computing power than non-preemptive algorithms.
2. **Starvation:** If short processes keep arriving, a long process might never finish (it keeps getting paused).

Preemptive Priority Scheduling

1. Introduction

This project implements the **Priority Preemptive Scheduling** algorithm. Unlike standard priority scheduling where a task runs to completion once started, this "Preemptive" version is more aggressive: if a more important task arrives while the CPU is busy, the CPU effectively pauses the current task to handle the VIP task immediately.

2. The Core Idea

The logic of this code is based on one golden rule: **Always work on the most important task available right now.**

How it works in Real Life: Imagine you are a customer service clerk helping a regular customer. Suddenly, your Manager (High Priority) walks up to your desk.

- **In Non-Preemptive:** You make the Manager wait until you finish helping the customer.

- **In Preemptive (This Code):** You apologize to the customer, pause their request, serve the Manager immediately, and only return to the customer once the Manager leaves.

3. Code Explanation

The Python code implements this using a **Time Step** approach. Here is the breakdown of the logic:

A. The Setup (rt and Priority) Just like in Shortest Job First, we need to track how much work is left for each process. We also need to extract the priority values.

B. The Main Loop (The Clock) The `while` loop acts as the system clock, incrementing time one unit at a time (`current_time += 1`). This step-by-step approach is crucial. Because high-priority processes can arrive at any random time, the system must re-evaluate the queue **every single second** to decide who should use the CPU next.

C. The Decision Logic Inside the loop, the code scans all processes to find the "VIP":

1. **Is it here?** (`at <= current_time`)
2. **Is it unfinished?** (`rt > 0`)
3. **Is it more important?** (`priority[i] < best_priority`)

If the code finds a process with a better (lower) priority score than the one currently running, it updates the `idx`. In the next line of execution, the CPU switches to this new process immediately.

4. Test Example (Dry Run)

To prove the code works, let's trace it with a simple example where **Lower Number = Higher Priority**.

Input:

- **P1:** Arrives at 0, Needs 6s, **Priority 10** (Low).
- **P2:** Arrives at 1, Needs 4s, **Priority 1** (High).

Execution Trace:

- **Time 0:** Only P1 is here. CPU starts P1.
    - *P1 Remaining: 5.*
- **Time 1:** P2 arrives with Priority 1.
    - **Compare:** P1 (Prio 10) vs P2 (Prio 1).
    - **Action:** P2 is more important. CPU **preempts** (pauses) P1 and switches to P2.
- **Time 2 - 5:** P2 runs uninterrupted because it has the highest priority.
- **Time 5:** P2 finishes. P2 is done.

- **Time 5:** CPU looks for work. Only P1 is left. CPU resumes P1.
- **Time 10:** P1 finishes remaining work.

Advantages of this Code:

- **Responsiveness:** Critical tasks (system errors, VIP requests) are handled immediately, making it ideal for Real-Time Operating Systems (RTOS).
- **Flexibility:** It adapts instantly to changing environments; a high-priority task never waits behind a low-priority one.

Limitations:

- **Starvation:** If high-priority processes keep arriving efficiently, a low-priority process might strictly wait forever (it never gets the CPU).
- **Context Switching Overhead:** Stopping a process, saving its state, and loading a new one takes computing power. Frequent switching can slow down the system.

Priority Non-Preemptive Scheduling

1. Introduction

This project implements the **Priority Non-Preemptive Scheduling** algorithm. Unlike the preemptive version, this algorithm respects the rule of "commitment": once the CPU starts working on a process, it will not stop until that process is completely finished, even if a more important process arrives in the meantime.

2. The Core Idea

The logic of this code is based on one golden rule: **Select the most important task available, and finish it completely.**

How it works in Real Life: Imagine you are a doctor treating a patient with a minor flu (Low Priority). Suddenly, a patient with a broken leg (High Priority) enters the waiting room.

- **In Preemptive:** You stop treating the flu patient immediately to handle the broken leg.
- **In Non-Preemptive (This Code):** You finish treating the flu patient first. The broken leg patient must wait in the lobby until you are free, regardless of how urgent their injury is.

3. Code Explanation

The Python code implements this using a **Process Selection** approach. Here is the breakdown of the logic:

A. The Setup (Tracking Completion) Unlike preemptive algorithms that track remaining time, here we track completion status.

B. The Main Loop (The Selector) The `while` loop continues until all processes are finished (`completed_count < n`). Inside this loop, the system looks at the queue to make a decision.

C. The Decision Logic The code scans for the best candidate based on three conditions:

1. **Is it here?** (`at[i] <= current_time`)
2. **Is it new?** (`not completed[i]`)
3. **Is it the most important?** (`priority[i] < best_priority`)

The Critical Difference (Non-Preemptive Logic): Once the best process (`idx`) is found, the code does not step forward by 1 second. Instead, it fast-forwards time by the **entire burst time** of that process:

This confirms the non-preemptive nature: the CPU is locked until the job is done.

4. Test Example (Dry Run)

To prove the code works, let's trace it with a simple example where **Lower Number = Higher Priority**.

Input:

- **P1:** Arrives at 0, Needs 5s, **Priority 10** (Low).
- **P2:** Arrives at 2, Needs 2s, **Priority 1** (High).

Execution Trace:

- **Time 0:** Only P1 is here. CPU picks P1.
    - *P1 starts execution.*
- **Time 2:** P2 arrives.
    - **Action:** Because this is **Non-Preemptive**, the CPU ignores P2 for now. It is busy with P1.
- **Time 5:** P1 finishes its 5 seconds.
    - *P1 Done. Current Time is 5.*
- **Time 5:** CPU becomes free. It looks at the queue. P2 is waiting.
    - CPU picks P2.
- **Time 7:** P2 finishes its 2 seconds. P2 Done.

Advantages of this Code:

- **Low Overhead:** Unlike preemptive scheduling, the CPU does not waste time frequently saving and loading process states (context switching).
- **Simplicity:** The logic is straightforward and ensures that once a task starts, it finishes without interruption.

Round Robin (RR) Scheduling

1. Introduction

This project implements the **Round Robin (RR)** scheduling algorithm. It is the most popular algorithm for time-sharing systems (like your laptop or phone) because it treats every process equally. It relies on a fixed time limit called a **Time Quantum** (or Time Slice).

2. The Core Idea

The logic of this code is based on one golden rule: **Fairness through Time Slicing.**

How it works in Real Life: Imagine you are playing a multiplayer video game on a single console with 3 friends, but there is only one controller.

- **The Rule (Quantum):** "Everyone gets to play for exactly 2 minutes."
- **The Process:**
    1. Friend A plays for 2 minutes. Time's up!
    2. Even if Friend A hasn't finished the level, they must pass the controller to Friend B and go to the back of the line.
    3. Friend B plays for 2 minutes, then passes to Friend C.
    4. Eventually, Friend A gets the controller back to finish their level.

3. Code Explanation

The Python code implements this using a **Queue-based** approach. Here is the breakdown of the logic:

A. The Setup (The Queue) Round Robin relies heavily on a First-In-First-Out (FIFO) queue.

We also track `rem_bt` (Remaining Burst Time) because processes are often paused and resumed multiple times.

B. The Decision Logic (Quantum Check) This block determines how much time passes:

4. Test Example (Dry Run)

To prove the code works, let's trace it. **Settings:** Time Quantum = **2 seconds**.

Input:

- **P1:** Arrives at 0, Needs 5s.
- **P2:** Arrives at 1, Needs 4s.

Execution Trace:

- **Time 0:** P1 is in queue. CPU picks **P1**.
  - Runs for 2s (Quantum). P1 Remaining: 3s.
  - *During this run (at Time 1), P2 arrived and joined the queue.*
  - **Queue now:** `[P2]`
  - P1 is not done, so it goes to the back. **Queue now:** `[P2, P1]`
- **Time 2:** CPU picks **P2**.
  - Runs for 2s. P2 Remaining: 2s.
  - **Queue now:** `[P1]`
  - P2 not done, goes to back. **Queue now:** `[P1, P2]`
- **Time 4:** CPU picks **P1**.
  - Runs for 2s. P1 Remaining: 1s.
  - **Queue now:** `[P2, P1]`
- **Time 6:** CPU picks **P2**.
  - Runs for 2s. P2 Remaining: 0s. **P2 Done.**
  - **Queue now:** `[P1]`
- **Time 8:** CPU picks **P1**.
  - Runs for 1s (Last bit). **P1 Done.**

Advantages of this Code:

- **No Starvation:** Every process, no matter how long, gets a turn every cycle.
- **Fairness:** No process can monopolize the CPU.

Limitations:

- **Context Switching Overhead:** If the `quantum` is too small (e.g., 1ms), the CPU spends more time switching between processes than actually working.
- **Becomes FCFS:** If the `quantum` is very large (e.g., 100s), the algorithm behaves exactly like First-Come-First-Serve, losing its advantages.

Multilevel Queue (MLQ) Scheduling

1. Introduction

This project implements the **Multilevel Queue (MLQ)** scheduling algorithm. Unlike simple algorithms that dump all processes into one big bucket, MLQ categorizes processes into different "classes" (e.g., System Processes vs. Student Processes) and assigns each class to its own specific queue.

2. The Core Idea

The logic of this code is based on one golden rule: **Class Hierarchy.**

How it works in Real Life: Imagine a bank with two lines:

- **Queue 1 (VIP):** Served by a dedicated manager.
- **Queue 2 (Regular):** Served by a standard teller.

The rule in this specific code is strict: **The Regular line is frozen until the VIP line is completely empty.** Even if a Regular customer arrived 1 hour ago, they cannot be served if there is a VIP customer present.

3. Code Explanation

The Python code implements this using a **Sequential Execution** approach. Here is the breakdown of the logic:

A. The Separation (Step 1) Before any scheduling happens, the code separates the single list of processes into distinct groups based on their queue number.

B. The Time Shift (The "Wait" Logic) This is the most critical logic in the code. Because Queue 2 cannot start until Queue 1 is finished, the code manually updates the "Arrival Time" of processes in Queue 2.

4. Test Example (Dry Run)

To prove the code works, let's trace it. **Setup:**

- **Queue 1 (High Priority):** Uses FCFS.
- **Queue 2 (Low Priority):** Uses Round Robin.

Input:

- **P1 (Queue 1):** Arrives at 0, Needs 5s.
- **P2 (Queue 2):** Arrives at 0, Needs 4s.

Execution Trace:

1. **Queue 1 starts:** The code sees P1 in Queue 1.
    - It runs P1 from Time 0 to 5.
    - `current_time` becomes 5.
2. **Queue 2 starts:** The code sees P2 in Queue 2.
    - P2 actually arrived at 0.
    - **The Shift:** But `current_time` is 5. So the code changes P2's arrival time to 5.
    - It passes P2 to the Round Robin algorithm.
    - Round Robin starts P2 at Time 5 (not 0).
    - P2 runs from 5 to 9.
3. **Finish:** Total time is 9.

5. Performance Analysis

The code calculates standard metrics by aggregating the results from the sub-algorithms.

Advantages of this Code:

- **Customization:** You can mix and match algorithms (e.g., use FCFS for system tasks and Round Robin for user tasks).
- **Priority Enforcement:** Critical system tasks (Queue 1) are guaranteed to finish before background tasks (Queue 2) even start.