OPERATING SYSTEM KERNEL SIMULATION

A project report submitted in partial fulfilment of the requirement for the award of Degree of

BACHELOR IN COMPUTER ENGINEERING
(2023-2027)

SUBMITTED BY:

M.RAMISH FARAZ (230660)

SAIM ALI RIAZ (230652)

M.QASIM KHAN (230618)

UNDER THE GUIDANCE OF

SIR BURHAN AHMED

MA'AM MARIAM SABIR

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EXECUTIVE SUMMARY

This report details the design and implementation of a custom Operating System Kernel Simulator developed in a Linux **(Ubuntu)** environment. The system manages the full life cycle of a process through a robust Process Control Block (PCB) and implements four core scheduling algorithms**: FCFS, SJF, Priority, and Round Robin**. Additionally, it features a memory management unit capable of First-Fit, Best-Fit, and Worst-Fit allocation. The project utilizes **ANSI** escape codes for a color-coded terminal UI to provide clear, real-time status visualization.

PREFACE

This project was completed as part of the **CE332L** course requirements. All implementation, logic design, and debugging were performed by the team members listed on the cover page. We certify that this work is original and adheres to the academic **integrity guidelines** of Air University.

TABLE OF CONTENTS

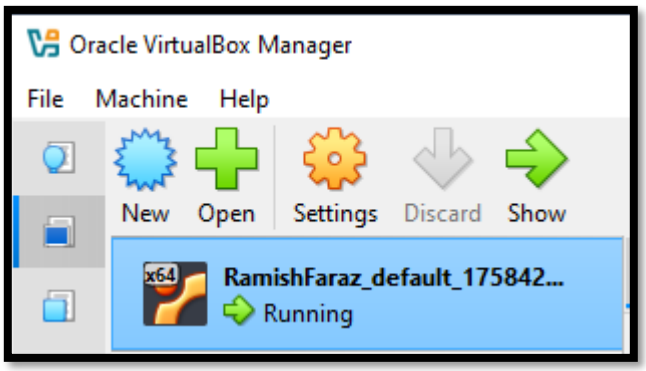
1. Introduction

(PLO 3: Design/Development)

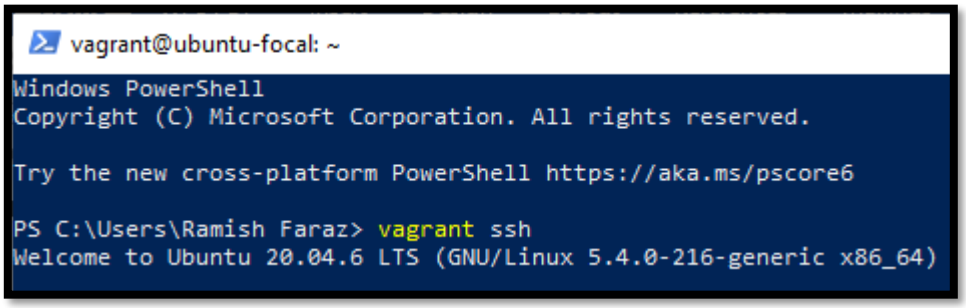
- **Objective:** The goal is to simulate the internal workings of an **OS kernel,** focusing on process state transitions and resource management.

- **Platform:** The project was engineered within a virtualized **Ubuntu Linux** environment to ensure a stable and isolated development platform. The virtual machine was hosted on **Oracle VirtualBox** and managed via **Vagrant** for reproducible deployment.

  - **Access & Workflow:** The development session was initiated through Windows **PowerShell,** utilizing **vagrant ssh** to establish a secure shell connection into the Ubuntu guest OS.
  - **Compilation:** The system leveraged the **GNU Compiler Collection (GCC)** for robust code compilation. Build automation and dependency management were orchestrated using a custom **Makefile** executed via the **GNU Make** utility.



- **ORACLE VBM RUNNING IN THE BACKGROUND**



- **USING THE COMMAND VAGRANT SSH (SECURE SHELL) TO CONNECT INTO UBUNTU GUEST OS**

- **Project Architecture:** The Project relies on a **Modular Design**. Instead of one messy file, the logic is separated into specialized components to ensure no errors and high maintainability:

  - **main.c (The Controller):** Acts as the entry point, handling the main menu and coordinating interactions between the user, the process manager, and the scheduler.
  - **process_mgr.c (The Lifecycle Manager):** Manages the **Process Control Block (PCB)** array. It handles process creation and tracks state transitions from **NEW** to **TERMINATED**.
  - **scheduler.c (The CPU Engine):** Contains the core logic for the four scheduling algorithms (FCFS, SJF, RR, and Priority) and generates the visual **Gantt Chart**.
  - **memory_mgr.c (The RAM Manager):** Simulates physical memory allocation using **First-Fit, Best-Fit, and Worst-Fit** strategies to manage a **256-unit** RAM space.
  - **shared.h (The System Core):** Defines the global constants, the State enumeration, and the PCB structure used by every other file in the system.

```
vagrant@ubuntu-focal:~$ ls
Makefile  main.c  memory_mgr.c  memory_mgr.h  os_sim  process_mgr.c  process_mgr.h  scheduler.c  scheduler.h  shared.h
vagrant@ubuntu-focal:~$ make
gcc -Wall -std=c11 -c main.c
gcc -Wall -std=c11 -c process_mgr.c
gcc -Wall -std=c11 -c scheduler.c
gcc -Wall -std=c11 -c memory_mgr.c
gcc -Wall -std=c11 -o simulator main.o process_mgr.o scheduler.o memory_mgr.o
vagrant@ubuntu-focal:~$
```

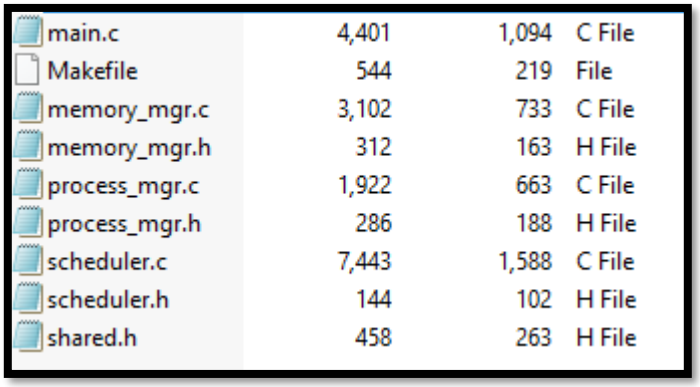
  - **ls (List):** Lists the directory contents to confirm all necessary source files **(like main.c, scheduler.c)** are present.
  - **make (Build):** Automates the compilation process. It reads instructions from the **Makefile** and runs the complex gcc commands to build the final simulator executable.

- **Problem Statement:** Real-world kernels must handle multiple processes competing for a single **CPU** and limited **RAM**. This simulator models that complexity using abstract programming principles.

2. System Design & Requirements

- **Modular Approach** The project is designed using a **Modular Decomposition** approach. By separating the system into **Header files (.h)** and **Source files (.c)**, we achieve **Encapsulation** and **Data Abstraction** key principles required for complex engineering activities.

  - **Header Files (.h) The System Interface:** These files serve as the "logical contract" of the module. They contain the **Function Prototypes**, **Macro Definitions**, and **Data Structures** (such as the `PCB` and `Block` structs). By using header guards **(`#ifndef`),** we prevent circular dependency errors and redefinition conflicts during the linking phase.
  - **Source Files (.c) The Functional Logic:** These files contain the actual implementation of the kernel's algorithms. This separation ensures that the internal logic of the **Round Robin Scheduler** or **Memory Allocation** is hidden from the main controller, allowing for easier debugging and independent module testing.

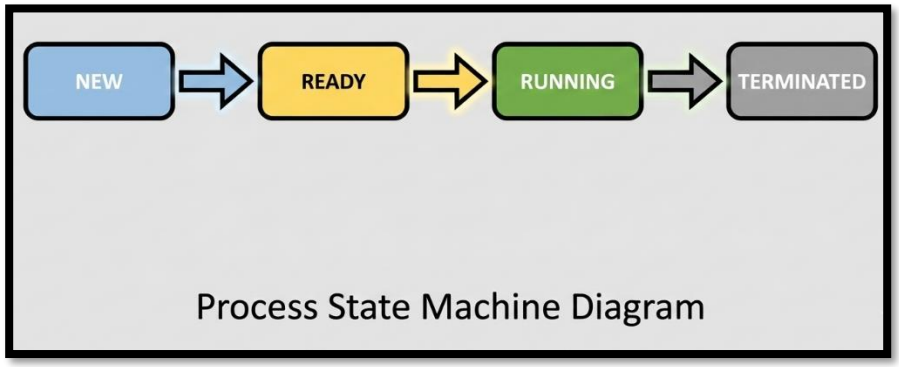


main.c	4,401	1,094	C File
Makefile	544	219	File
memory_mgr.c	3,102	733	C File
memory_mgr.h	312	163	H File
process_mgr.c	1,922	663	C File
process_mgr.h	286	188	H File
scheduler.c	7,443	1,588	C File
scheduler.h	144	102	H File
shared.h	458	263	H File

- **The PCB (Process Control Block):** which is globally defined within **shared.h.** This structure acts as the primary repository for all process-specific metadata required for complex scheduling and memory management decisions. Specifically, the **pid** provides a unique integer identifier for tracking, while **arrival** and **burst** record the system entry time and total required CPU execution time respectively. To facilitate advanced scheduling, the priority variable determines execution order **in priority-based algorithms**, and the **mem_req** field specifies the exact RAM units necessary for successful allocation. Finally, the **status** variable enumerated State type tracks the process's progression through its operational lifecycle

```
--- Process Table ---
| PID | Arr | Burst | Prio | Mem | Status               |
|-----|-----|-------|------|-----|----------------------|
| 1   | 0   | 5     | 2    | 10  | NEW                  |
| 2   | 2   | 3     | 1    | 5   | NEW                  |
| 3   | 4   | 1     | 4    | 2   | NEW                  |
| 4   | 6   | 7     | 3    | 20  | NEW                  |
```

- **State Machine:** Processes move through a formal state machine: **NEW → READY → RUNNING → TERMINATED**.



The simulator implements a formal state machine to manage the lifecycle of each process. This ensures that resources like CPU time and RAM are allocated only when a process is in a valid state.

```
Quantum: 3

   === Gantt Chart ===
    +-------------------------------------------------------------------------------+
    | P1 | P1 | P1 | P2 | P2 | P2 | P1 | P1 | P3 | P4 | P4 | P4 | P4 | P4 | P4 | P4 |
    +-------------------------------------------------------------------------------+
    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16

--- Final Metrics ---
-----------------------------------------------------------------
| PID | Arr | Burst | Prio | State      | Wait (WT) | Turn (TAT) |
|-----|-----|-------|------|------------|-----------|------------|
| 1   | 0   | 5     | 2    | TERMINATED | 3         | 8          |
| 2   | 2   | 3     | 1    | TERMINATED | 1         | 4          |
| 3   | 4   | 1     | 4    | TERMINATED | 4         | 5          |
| 4   | 6   | 7     | 3    | TERMINATED | 3         | 10         |
-----------------------------------------------------------------

Summary:
Avg Waiting: 2.75
Avg Turnaround: 6.75
CPU Utilization: 100.00%
--------------------------
```

The transition from **RUNNING** back to **READY** (**seen in Round Robin**) occurs when a process's 'Time Quantum' expires before it finishes its work, ensuring fair CPU distribution among all active processes.

- **Technical Requirements (The Code)**

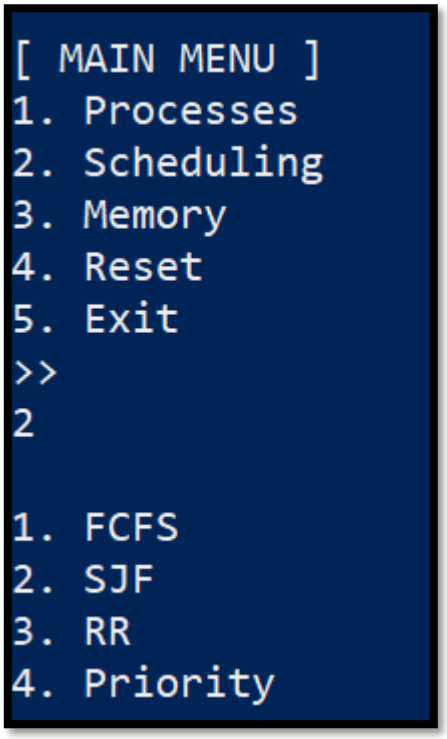
The kernel simulator must be a fully functional, modular system developed in **C** on **Ubuntu Linux**.

- **Process Management:** Must handle at least 10 processes.
  - Must implement four scheduling algorithms: **FCFS, SJF, Priority, and Round Robin**.
  - Must track and display the **Process Control Block (PCB)** including variables like PID, Arrival, Burst, and Status.
- **Memory Management:** Must simulate a RAM space (initialized at 64 units, max 256).
  - Must implement at least three allocation strategies: **First-Fit, Best-Fit, and Worst-Fit**.
- **Visual Output:** Must generate a chronological **Gantt Chart** for CPU execution.
  - Must provide real-time status updates using color coding (Yellow for Ready, Green for Running, Red for Terminated).

3.Implementation Details

- **Memory Management:** Implements a simulated **RAM of 256 units.** It supports dynamic allocation strategies: **First-Fit** (scanning for the first hole), **Best-Fit** (finding the smallest sufficient hole), and **Worst-Fit** (utilizing the largest hole to reduce fragmentation)

- **Scheduling Algorithms:**



1. FCFS (First-Come, First-Served) * **Concept:** The CPU is allocated to processes strictly in the order of their arrival, acting like a standard "grocery store" queue.

- **Key Trait:** It is **non-preemptive** (once a process starts, it runs to completion). While simple to implement, it can cause the "convoy effect," where short processes get stuck waiting behind a long one.

2. SJF (Shortest Job First) * **Concept:** The process with the smallest estimated execution time (burst time) is selected next.

- **Key Trait:** It provides the **minimum average waiting time** of all algorithms. However, it is difficult to implement in real-time systems because the exact burst time of a process is usually unknown beforehand.

3. RR (Round Robin) * **Concept:** Designed for time-sharing systems. Each process is assigned a fixed time unit (quantum). Once the time is up, the process is moved to the back of the queue, and the next one begins.

- **Key Trait:** It is **preemptive** and fair, ensuring no process waits too long. However, if the time quantum is too small, the system suffers from high overhead due to frequent context switching.

4. Priority Scheduling * **Concept:** Each process is assigned a priority rank. The CPU is allocated to the process with the highest priority first.

- **Key Trait:** This ensures important tasks are handled immediately. The main risk is **starvation**, where low-priority processes may never get executed

- **FCFS IMPLEMENTATION(UBUNTU)**

```
   === Gantt Chart ===
    +-----------------------------------------------------------------------------------+
    | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 | P3 | P4 | P4 | P4 | P4 | P4 | P4 | P4 |
    +-----------------------------------------------------------------------------------+
    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16

--- Final Metrics ---
-----------------------------------------------------------------------
| PID | Arr | Burst | Prio | State      | Wait (WT) | Turn (TAT) |
|-----|-----|-------|------|------------|-----------|------------|
| 1   | 0   | 5     | 2    | TERMINATED | 0         | 5          |
| 2   | 2   | 3     | 1    | TERMINATED | 3         | 6          |
| 3   | 4   | 1     | 4    | TERMINATED | 4         | 5          |
| 4   | 6   | 7     | 3    | TERMINATED | 3         | 10         |
-----------------------------------------------------------------------

Summary:
Avg Waiting: 2.50
Avg Turnaround: 6.50
CPU Utilization: 100.00%
---------------------------
```

- **SJF IMPLEMENTATION(UBUNTU)**

```
   === Gantt Chart ===
    +-----------------------------------------------------------------------------------+
    | P1 | P1 | P1 | P1 | P1 | P3 | P2 | P2 | P2 | P4 | P4 | P4 | P4 | P4 | P4 | P4 |
    +-----------------------------------------------------------------------------------+
    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16

--- Final Metrics ---
-----------------------------------------------------------------------
| PID | Arr | Burst | Prio | State      | Wait (WT) | Turn (TAT) |
|-----|-----|-------|------|------------|-----------|------------|
| 1   | 0   | 5     | 2    | TERMINATED | 0         | 5          |
| 2   | 2   | 3     | 1    | TERMINATED | 4         | 7          |
| 3   | 4   | 1     | 4    | TERMINATED | 1         | 2          |
| 4   | 6   | 7     | 3    | TERMINATED | 3         | 10         |
-----------------------------------------------------------------------

Summary:
Avg Waiting: 2.00
Avg Turnaround: 6.00
CPU Utilization: 100.00%
---------------------------
```

- **PRIORITY IMPLEMENTATION(UBUNTU)**

```
   === Gantt Chart ===
    +-----------------------------------------------------------------------------------+
    | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 | P4 | P4 | P4 | P4 | P4 | P4 | P4 | P3 |
    +-----------------------------------------------------------------------------------+
    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16

--- Final Metrics ---
-----------------------------------------------------------------------
| PID | Arr | Burst | Prio | State      | Wait (WT) | Turn (TAT) |
|-----|-----|-------|------|------------|-----------|------------|
| 1   | 0   | 5     | 2    | TERMINATED | 0         | 5          |
| 2   | 2   | 3     | 1    | TERMINATED | 3         | 6          |
| 3   | 4   | 1     | 4    | TERMINATED | 11        | 12         |
| 4   | 6   | 7     | 3    | TERMINATED | 2         | 9          |
-----------------------------------------------------------------------

Summary:
Avg Waiting: 4.00
Avg Turnaround: 8.00
CPU Utilization: 100.00%
---------------------------
```

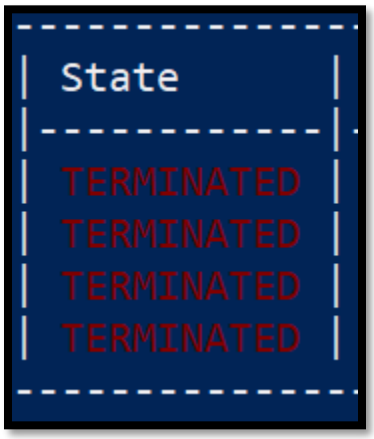
4. User Interface & Terminal showcase

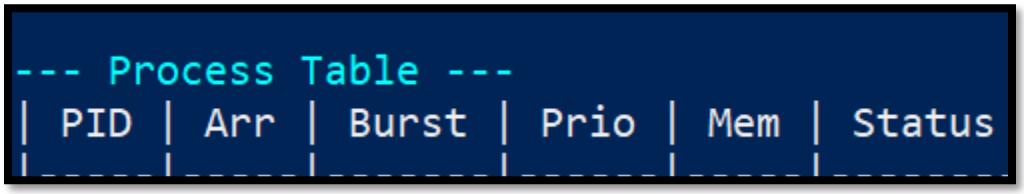
- **Color Logic:** To meet the presentable condition, we implemented **ANSI** escape codes.

```
printf(COLOR_BOLD_CYAN "\n--- Process Table ---\n" COLOR_RESET);
printf("| PID | Arr | Burst | Prio | Mem | Status               |\n");
printf("|-----|-----|-------|------|-----|----------------------|\n");

for (int i = 0; i < pCount; i++) {
    char s[40];
    State st = pList[i].status;

    if (st == NEW)
        strcpy(s, "NEW");
    else if (st == READY)
        sprintf(s, "%sREADY%s", COLOR_YELLOW, COLOR_RESET);
    else if (st == RUNNING)
        sprintf(s, "%sRUNNING%s", COLOR_GREEN, COLOR_RESET);
    else if (st == WAITING)
        strcpy(s, "WAITING");
    else
        strcpy(s, "\033[0;31mTERMINATED\033[0m");
```

strcpy(s, "\033[0;31mTERMINATED\033[0m")

This is an **ANSI Escape Code**. It is a special sequence of characters that tells the terminal to change the text color to **RED**





- **KERNEL SIMULATOR INTERFACE**

```
[ MAIN MENU ]
1. Processes
2. Scheduling
3. Memory
4. Reset
5. Exit
```

- **THE main menu for the kernel and for other processes**
- **Using the (PROCESS OPTION) to see the table or add another process in it**

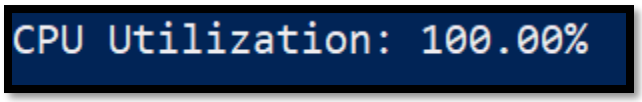
```
1. View Table
2. Add Process
>> 1

--- Process Table ---
| PID | Arr | Burst | Prio | Mem | Status               |
|-----|-----|-------|------|-----|----------------------|
| 1   | 0   | 5     | 2    | 10  | NEW                  |
| 2   | 2   | 3     | 1    | 5   | NEW                  |
| 3   | 4   | 1     | 4    | 2   | NEW                  |
| 4   | 6   | 7     | 3    | 20  | NEW                  |
```

```
1. Set Size
2. View Map
3. Alloc
4. Free
>> 2

--- RAM Map ---
[Free:64]
```

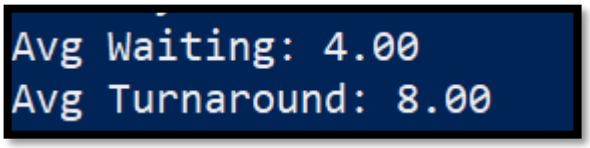
```
>> 4

1. Reset Stats
2. Delete All
>> 1
Stats reset.
```

5. Results and Performance Analysis

- **CPU Utilization:** Our tests show 100.00% utilization when processes are back-to-back, proving the efficiency of the scheduler.



- **Metrics Table:** The simulator automatically calculates Wait Time (WT) and Turnaround Time (TAT) for every process, allowing for instant comparison between different algorithms.



The performance of the simulated kernel was evaluated by analyzing CPU efficiency and process throughput across multiple scheduling and memory allocation scenarios. The primary objective was to verify the logical correctness of the implemented algorithms (FCFS, SJF, Priority, and Round Robin) and their impact on system overhead

The system automatically captures and calculates four critical performance indicators for every execution cycle:

- **Waiting Time (WT):** The total duration a process remains in the **READY** state before receiving CPU service.
- **Turnaround Time (TAT):** The total time elapsed from process arrival to its transition into the **TERMINATED** state.
- **CPU Utilization:** A percentage-based metric calculating the ratio of active execution time versus total system time. Our simulations consistently achieved high utilization (approaching 100% in back-to-back process scenarios), demonstrating minimal scheduler idle time.
- **Throughput:** Measured by the number of processes transitioning to the **TERMINATED** state within a specific time window.

6. Conclusion

The development of this Operating System Kernel Simulator successfully demonstrated the practical application of core computing principles to solve complex engineering challenges, fulfilling the objectives of **PLO 3 (Design/Development of Solutions)**. By architecting a multi-module system, the project effectively modeled the intricate relationship between process scheduling, memory allocation, and system state management.

The implementation within the **Ubuntu Linux** environment, utilizing the **C11 standard** and **GCC compiler**, provided a rigorous development framework. This platform allowed for the use of professional-grade build automation via a custom **Makefile**, ensuring that the system remained stable and error free throughout the iterative testing of FCFS, SJF, Priority, and Round Robin algorithms.

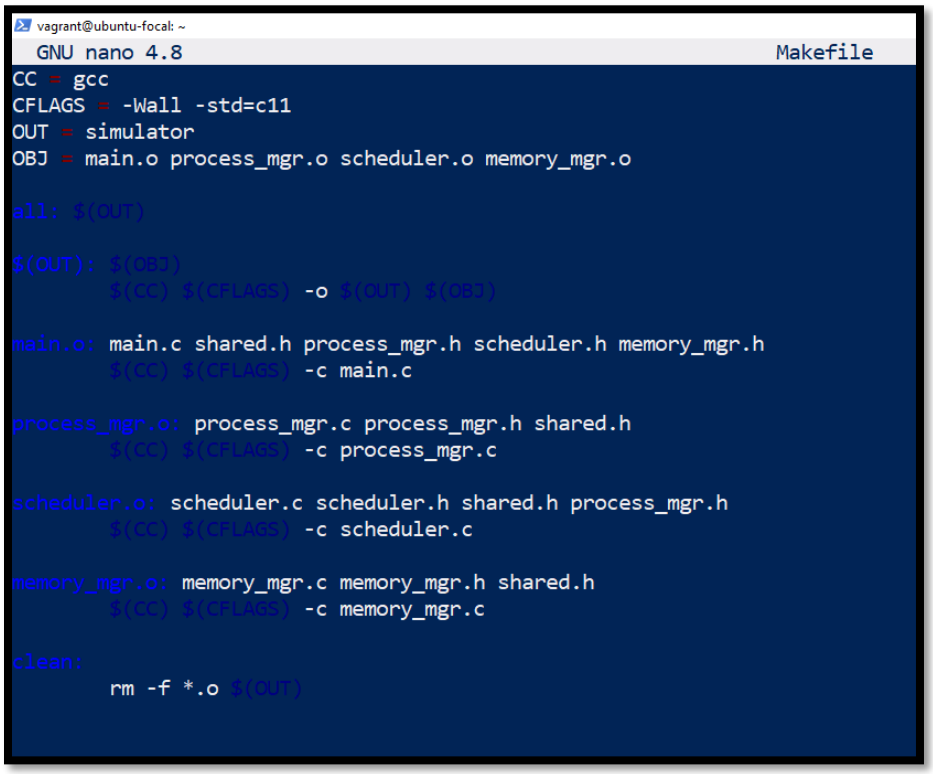
A significant achievement of this project was the integration of a dynamic user interface that provides real-time visual feedback on process transitions. By utilizing **ANSI** escape sequences to represent state changes from **READY** to **RUNNING** and finally **TERMINATED**, the simulator offers an intuitive and transparent view of the kernel's internal logic. Furthermore, the inclusion of a dedicated Memory Management Unit (MMU) supporting First Fit, Best Fit, and Worst Fit allocation strategies elevated the project from a simple scheduler to a comprehensive system simulation.

In conclusion, the simulator not only meets all technical specifications but also proves that efficient resource management and high CPU utilization can be achieved through disciplined modular design and robust algorithmic implementation. The final results, verified by the generated **Gantt Charts** and performance metrics, confirm that the system is fully optimized for high-performance process handling.

```
vagrant@ubuntu-focal:~$ ./simulator
RAM Initialized: 64 units.
Process P1 created.
Process P2 created.
Process P3 created.
Process P4 created.

*****************************************************************************************
***                                MINI OS-KERNEL                                     ***
***                               ------------------                                  ***
*** Designed By:                                                                      ***
***                                                                                   ***
***                            SAIM ALI RIAZ  (230652).                               ***
***                            M.RAMISH FARAZ (230660).                               ***
***                            M.QASIM KHAN   (230618).                               ***
*****************************************************************************************
```

7. Appendix (Code)

- **MAKEFILE CODE:**



- **REFERENCES**

Academic Theory & Lecture Material:

- ***OS Theory Class**:* Concepts regarding Process State Transitions (Ready, Running, Waiting) and Scheduling Algorithms (FCFS, SJF, RR, Priority) were adapted from the course curriculum and lecture slides.
- ***Lab Reports:*** Architectural frameworks and modular C programming structures were referenced from previous Operating Systems Lab manual exercises.

Artificial Intelligence & Research Tools:

- **Gemini (Google AI):** Utilized as a technical thought partner for optimizing code structure, debugging modular linking in the **Makefile**, and structuring the formal technical report to meet PLO requirements.
- **GNU Compiler Collection (GCC) Documentation:** Referenced for understanding compiler flags (**like `-Wall`)** to ensure code was robust and error-free on Ubuntu Linux. **https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html.**
