Final Project Report Of Operating System Mini Simulator

Project Name: Design and Implementation of Mini OS Simulator

Submitted to: Fahad Irshad (Lab Engineer)

Submitted By:

Name (ID) : Asfand Amjad (F2023266864)

Section : V7

Session

Fall 2025

Design and Implementation of a Mini Operating System Simulator

Acknowledgement

I would like to express my sincere gratitude to **Sir Fahad Irshad**, Instructor of the Operating Systems Lab (Section V7), for his valuable guidance, continuous support, and constructive feedback throughout the completion of this Complex Computing Problem (CCP).

His clear explanation of operating system concepts and emphasis on understanding system-level design greatly contributed to the successful development of this Mini Operating System Simulator. This project would not have been possible without his mentorship and encouragement.

Table of Contents

Design and Implementation of a Mini Operating System Simulator

1. Introduction

An operating system is the core software component responsible for managing hardware resources and providing essential services to application programs. Key responsibilities of an operating system include **process management, CPU scheduling, synchronization, and resource allocation**, all of which must function correctly in a concurrent environment.

The objective of this Complex Computing Problem (CCP) is to design and implement a **Mini Operating System Simulator** that models these responsibilities in a controlled, educational setting. Instead of developing a real kernel, this simulator focuses on *how an operating system thinks and makes decisions*. The system integrates multiple operating system concepts into a single cohesive program, ensuring that no module operates in isolation.

This project has been implemented using **C++**, leveraging multithreading, synchronization primitives, and structured data models. The simulator demonstrates how processes are created, scheduled, synchronized, and safely executed while preventing deadlock through resource management techniques.

2. Overall System Architecture

The Mini OS Simulator is divided into three logically independent but tightly integrated modules:

1. **Process Management and Scheduling Module**
2. **Producer–Consumer Synchronization Module**
3. **Resource Management and Deadlock Prevention Module**

Each module communicates through shared data structures protected by synchronization mechanisms. This modular architecture closely resembles the layered design used in real operating systems.

3. Process Control Block (PCB) Design

The **Process Control Block (PCB)** is the fundamental data structure used to represent a process within the simulator. Every process generated by the system is associated with exactly one PCB.

3.1 PCB Attributes

- **Process ID (PID):** A unique identifier assigned sequentially to each process.
- **Arrival Time:** Records the system time at which the process enters the ready queue.
- **CPU Burst Time:** Represents the amount of CPU time required by the process.
- **Priority:** Used to determine the importance of the process during scheduling.
- **Maximum Resource Need Vector:** Specifies the maximum number of instances of each resource required.
- **Allocation Vector:** Tracks resources currently allocated to the process.
- **Process State:** Indicates the current lifecycle stage of the process.

3.2 Process States

The simulator models the classical operating system process states:

- **NEW:** Process has been created but not yet scheduled.
- **READY:** Process is waiting in the ready queue for CPU allocation.
- **RUNNING:** Process is currently executing on the CPU.
- **BLOCKED:** Process cannot proceed due to unsafe resource allocation.
- **TERMINATED:** Process has completed execution and released resources.

These states allow accurate simulation of process transitions and system behavior.

4. Producer–Consumer Module

4.1 Objective

The producer–consumer module simulates the concurrent arrival of processes into the operating system. This reflects real-world systems where multiple applications and users create processes simultaneously.

4.2 Producer Threads

The simulator implements **two producer threads**, each responsible for generating new processes at runtime. Each producer:

- Creates a new PCB
- Assigns random burst time and priority values
- Generates a maximum resource requirement vector
- Inserts the process into the ready queue

To prevent overflow, the ready queue is implemented as a **bounded buffer** with a fixed size.

4.3 Consumer Thread (CPU)

The consumer thread represents the **CPU**. It removes processes from the ready queue and forwards them for execution after performing safety checks.

4.4 Synchronization Strategy

Proper synchronization is achieved using:

- **Mutex (std::mutex)** for mutual exclusion
- **Condition Variables** to handle full and empty buffer conditions

Busy waiting is strictly avoided. Producers are blocked when the ready queue is full, and the consumer is blocked when the queue is empty. This ensures efficient CPU usage and correct concurrent behavior.

5. CPU Scheduling and Execution Model

Once a process is selected by the consumer thread, it is scheduled for execution based on its availability and resource safety.

5.1 Scheduling Behavior

- Processes are fetched from the ready queue in arrival order
- State transitions occur as READY → RUNNING → TERMINATED
- CPU execution time is simulated using sleep intervals equal to burst time

This approach provides a realistic abstraction of CPU execution while remaining suitable for a lab-level simulation.

6. Resource Management and Banker's Algorithm

6.1 Need for Deadlock Prevention

In a multiprogramming environment, improper allocation of limited resources can lead to deadlock. To address this, the simulator incorporates a simplified implementation of **Banker's Algorithm**.

6.2 Resource Model

The system maintains the following data structures:

- **Available Vector:** Number of free instances for each resource type
- **Max Need Matrix:** Maximum demand of each process
- **Allocation Matrix:** Currently allocated resources

6.3 Safety Check Mechanism

Before a process is allowed to execute, the system evaluates whether allocating its remaining required resources will keep the system in a safe state.

A process is considered safe if:

$(\text{MaxNeed} - \text{Allocation}) \leq \text{Available}$

6.4 Blocking and Recovery

- If the state is unsafe, the process is moved to the blocked queue
- Upon resource release, blocked processes are re-evaluated
- Safe processes are reinserted into the ready queue

This mechanism ensures that the system never enters an unsafe or deadlocked state.

7. User Interface and Interaction

The simulator provides a menu-driven interface for user interaction:

1. **Start Simulation** – Launches producer and consumer threads
2. **Add Process Manually** – Allows user-defined process creation
3. **Display System State** – Shows ready queue, blocked queue, and available resources
4. **Exit** – Safely terminates the simulation

This interface simplifies testing and demonstration of system behavior.

8. Limitations of the Simulator

Despite accurately modeling core operating system concepts, the simulator has the following limitations:

- Command-line based interface only
- Simplified scheduling strategy
- No true preemption or time slicing
- Fixed number of resource types
- No memory management or I/O scheduling

These constraints keep the project aligned with undergraduate lab requirements.

9. Future Scope and Extensions

The Mini OS Simulator can be extended into a Final Year Project by incorporating:

- Advanced scheduling algorithms (Round Robin, Priority with Preemption)
- Graphical visualization of process execution
- Distributed resource management
- Cloud workload simulation
- Real-time operating system support

10. Conclusion

This Complex Computing Problem demonstrates a comprehensive understanding of operating system fundamentals by integrating process management, synchronization, CPU scheduling, and deadlock prevention into a single simulator.

The project emphasizes correct design, modularity, and safe concurrent execution, reflecting real operating system behavior. The Mini OS Simulator successfully fulfills the objectives of the Operating Systems Lab and provides a strong foundation for advanced system-level projects.

THE END