Operating Systems Lab

Project Report

Submitted by : Muhammad Junaid

Submitted to : Muhammad Fahad

Student id : F2023266884

Section : V7

School of System and Technology

University of Management and Technology

Mini OS Simulator: Technical Report

1. Executive Summary

The **Mini OS Simulator** is an integrated C++ application designed to model the core functions of an operating system kernel. It simulates three critical subsystems: **Process Scheduling**, **Concurrent Task Management (Producer-Consumer)**, and **Deadlock Prevention (Banker's Algorithm)**. By utilizing POSIX threads and synchronization primitives, the system provides a realistic environment for observing how a kernel manages resources and schedules tasks in real-time.

2. System Architecture & Modules

The project is built using a modular approach to ensure scalability and ease of debugging.

A. Process & Scheduler Module

This module handles the execution of processes residing in the Ready Queue.

- **Dynamic Algorithm Switching**: The system evaluates the system load and chooses the most efficient algorithm:
    - **Priority Scheduling (Non-preemptive)**: Used when the Ready Queue has **5 or fewer** processes. This ensures that high-priority tasks are handled quickly in a low-load environment.
    - **Round Robin (RR) Scheduling (Preemptive)**: Used when the queue exceeds **5 processes**. It uses a **Time Quantum of 3 units** to prevent process starvation and ensure fair CPU time distribution.
- **Performance Metrics**: After execution, the module calculates and displays Waiting Time, Turnaround Time (TAT), and Average Statistics.

B. Producer-Consumer Module

This module manages the entry of tasks into the system using a multithreaded architecture.

- **Producers**: Two independent threads generate processes with randomized attributes (Burst Time, Priority, and Resource Needs) and attempt to place them into a **Bounded Buffer** of size 5.
- **Consumer**: A single thread acts as the "Middleman." it retrieves processes from the buffer, subjects them to a safety check, and moves them to the appropriate queue (Ready or Blocked).

C. Resource Management & Deadlock Prevention

This module acts as the system's "Safety Officer" using the **Banker's Algorithm**.

- **Pre-allocation Check**: Before a process is allowed to enter the Ready Queue, the system simulates the allocation of its maximum requested resources.

- **Safety Logic**: If the system cannot guarantee a "Safe Sequence" where all processes can eventually finish, the requesting process is moved to a **Blocked Queue** to prevent a potential deadlock.

3. Synchronization & Thread Safety

Concurrency is managed through strictly defined synchronization primitives to prevent race conditions and busy-waiting.

Primitive	**Role in Simulator**
empty_slots (Semaphore)	Tracks available space in the buffer. Producers wait here if the buffer is full.
full_slots (Semaphore)	Tracks occupied spaces in the buffer. The consumer waits here if the buffer is empty.
mutex_lock (Mutex)	Ensures that only one thread (Producer 1, Producer 2, or Consumer) can modify the buffer or queues at any given time.

4. Implementation Details

Data Structures

The core of the simulator is the Process struct, which tracks:

- **Execution Data**: PID, Burst Time, and Remaining Time.
- **Resource Matrices**: max_need, allocated, and need arrays (supporting 3 resource types: R0, R1, R2).

Integrated Simulation Flow

1. **System Initialization**: Resources are set (e.g., 10 units of R0, 5 of R1, 7 of R2).
2. **Concurrency Start**: Worker threads (Producers/Consumer) begin operating in the background.
3. **Kernel Menu**: The user interacts with the "Live" system to view the state, run the scheduler, or force-inject specific processes.

5. Limitations & Future Scope

Current Limitations

- **Non-Preemptive Priority**: While Round Robin is preemptive, the Priority Scheduling implementation waits for a process to finish its entire burst.
- **Static Resource Types**: The number of resource types is hardcoded to 3.
- **Memory Simulation**: The simulator lacks a Virtual Memory/Paging module.

Future Expansion (FYP Scope)

- Developing a **GUI-based Dashboard** using Qt or Dear ImGui.
- Implementing **Cloud Workload Scheduling** to simulate distributed resource management.
- Adding a **Memory Management Unit (MMU)** to simulate paging and segmentation.

6. Conclusion

The Mini OS Simulator successfully demonstrates the integration of complex OS concepts into a working, multithreaded system. By preventing deadlocks and dynamically adapting scheduling strategies, the project reflects the real-world logic used in modern operating system kernels.

Project Repository: https://github.com/junicoder/Mini-OS-Simulator