Mini Operating System Simulator

Complex Computing Problem (CCP) - Project Report

Author: ABDULWASAY

ID/Reg No: F2023266782

Date: 30/1/2026

Course: Operating Systems

Project: Integrated Process Management System

Table of Contents

9. Conclusion

1. Introduction

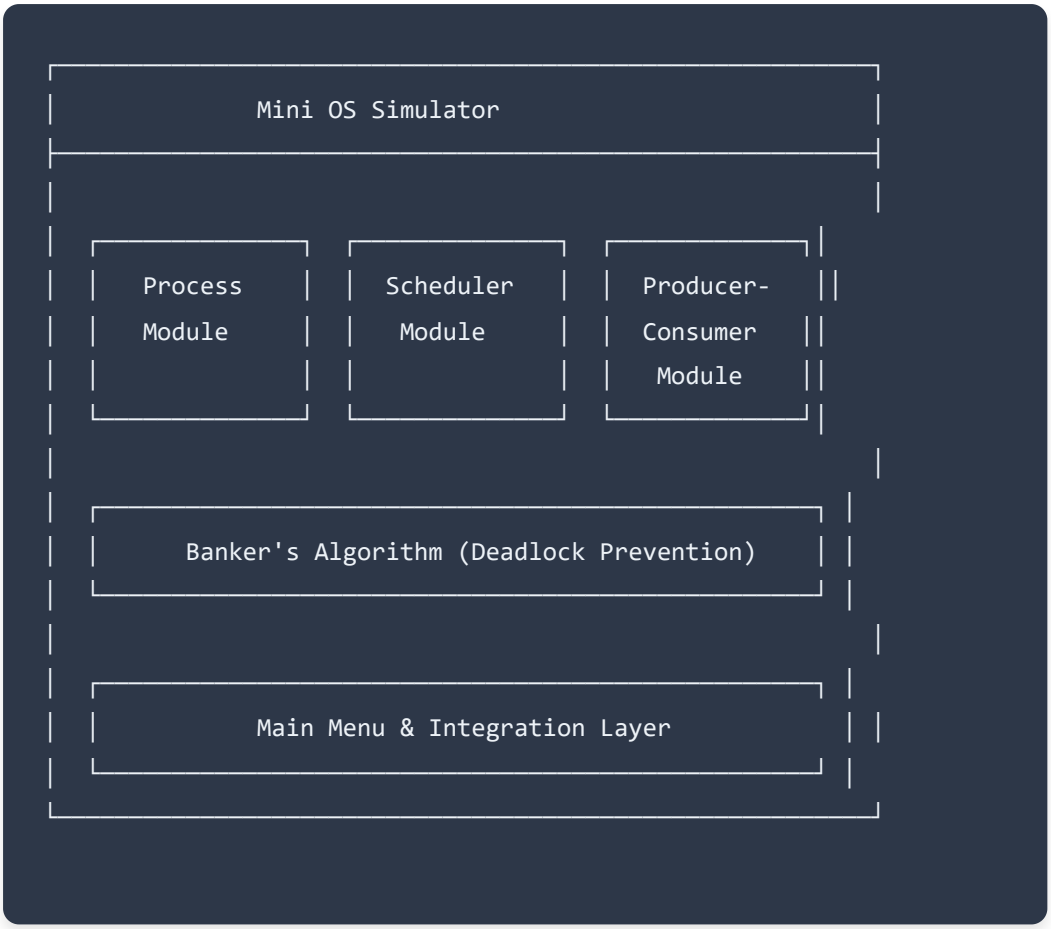
This report documents the design and implementation of a **Mini Operating System Simulator** that integrates three core OS concepts: Process Scheduling, Producer-Consumer synchronization, and Deadlock Prevention using Banker's Algorithm. The simulator demonstrates how an operating system manages processes, allocates resources, and prevents unsafe states in a concurrent environment.

Objectives

- Simulate CPU scheduling using **Priority** and **Round Robin** algorithms.
- Implement concurrent process generation and consumption using **semaphores**.
- Prevent deadlocks using **Banker's Algorithm** for resource allocation.
- Provide an integrated system demonstrating real OS behavior.

2. System Architecture

The simulator is built as a modular system consisting of four main components:



Component Overview

1. **Process Module**: Defines process structure with all required attributes.
2. **Scheduler Module**: Implements Priority and Round Robin scheduling algorithms.
3. **Producer-Consumer Module**: Handles concurrent process generation and consumption.
4. **Banker's Algorithm Module**: Manages resource allocation and deadlock prevention.

3. Module-wise Explanation

3.1 Process Module

File: `process.h`, `process.cpp`

The Process module defines the fundamental data structure representing a process in the system.

Key Attributes:

- **Process ID**: Unique identifier for each process.
- **Arrival Time**: Time when process enters the system.
- **Burst Time**: CPU time required for execution.
- **Priority**: Priority level (higher number = higher priority).
- **Resource Requirement Vector**: Array specifying required resources (R1, R2, ...).
- **Scheduling Statistics**: Waiting time, turnaround time, completion time.
- **Status Flags**: `isCompleted`, `isBlocked` for state tracking.

3.2 Scheduler Module

File: `scheduler.h`, `scheduler.cpp`

The Scheduler module implements two classical CPU scheduling algorithms with dynamic selection based on system load.

3.2.1 Priority Scheduling (Non-preemptive)

Algorithm: Priority Scheduling selects the process with the highest priority number from the ready queue. Once selected, a process runs to completion without preemption.

- Processes sorted by priority (higher number = higher priority).
- Non-preemptive: Process runs until completion.
- Handles arrival times correctly.

3.2.2 Round Robin Scheduling (Preemptive)

Algorithm: Round Robin uses a time quantum (2 units) to preemptively schedule processes in a circular fashion.

- Time quantum = 2 units (configurable).
- Preemptive: Process interrupted after quantum expires.
- Fair scheduling: All processes get equal CPU time.

3.2.3 Scheduler Factory

Dynamic Selection Rule:

Condition	Scheduler Used
≤ 5 ready processes	Priority Scheduling
> 5 ready processes	Round Robin Scheduling

3.3 Producer-Consumer Module

File: `producer_consumer.h`, `producer_consumer.cpp`

This module implements the classic Producer-Consumer pattern with proper synchronization to simulate concurrent process generation.

3.3.1 Bounded Buffer

Purpose: Acts as a shared buffer between producers and consumers with fixed capacity (size = 10).

Synchronization Mechanisms:

1. **Semaphores:**
   - `emptySlots`: Counts available empty slots (initialized to buffer size).
   - `fullSlots`: Counts filled slots (initialized to 0).
2. **Mutex:**
   - `mtx`: Protects critical sections (buffer access).

Producer Operation:

```
sem_wait(&emptySlots);       // Wait for empty slot (blocks if f
lock_guard<mutex> lock(mtx); // Enter critical section
buffer.push(process);        // Add process to buffer
sem_post(&fullSlots);        // Signal one more full slot
```

Consumer Operation:

```
sem_wait(&fullSlots);        // Wait for full slot (blocks if er
lock_guard<mutex> lock(mtx); // Enter critical section
process = buffer.front();    // Remove process from buffer
buffer.pop();
sem_post(&emptySlots);       // Signal one more empty slot
```

3.4 Banker's Algorithm Module

File: `banker.h`, `banker.cpp`

This module implements a simplified Banker's Algorithm to prevent deadlocks by ensuring the system never enters an unsafe state.

Data Structures:

1. **Available Vector**: Current available resources [R1, R2, ...].
2. **Max Matrix**: Maximum resource demand for each process.
3. **Allocation Matrix**: Currently allocated resources per process.
4. **Need Matrix**: Remaining resource need (Need = Max - Allocation).

4. Scheduling Decisions

4.1 Scheduler Selection Logic

The system uses a **dynamic scheduler selection** mechanism based on the number of ready processes:

Condition	Scheduler Used	Rationale
≤ 5 processes	Priority Scheduling	Efficient for small workloads, minimizes context switches.
> 5 processes	Round Robin	Ensures fairness, prevents starvation, better for larger workloads.

5. Semaphore Usage Explanation

5.1 Semaphore Types

The system uses **counting semaphores** to manage buffer synchronization:

1. `emptySlots` **Semaphore:**
   - Initial value: Buffer size (10).
   - Decremented: When producer adds item (`sem_wait`).
   - Incremented: When consumer removes item (`sem_post`).
2. `fullSlots` **Semaphore:**
   - Initial value: 0.
   - Decremented: When consumer removes item (`sem_wait`).
   - Incremented: When producer adds item (`sem_post`).

5.2 Synchronization Flow

Why Semaphores?

- **No Busy Waiting**: Threads block when waiting, saving CPU cycles.
- **Efficient**: Kernel-level blocking is more efficient than polling.
- **Prevents Race Conditions**: Ensures proper synchronization.

6. Deadlock Prevention Logic

6.1 Banker's Algorithm Overview

The Banker's Algorithm prevents deadlocks by ensuring the system **never enters an unsafe state**. An unsafe state is one where deadlock is possible, even if it hasn't occurred yet.

6.2 Safety Check Before Execution

Critical Rule: Before a process executes, the system checks if granting its requested resources would keep the system in a safe state.

```
Safety Algorithm Logic:
1. Initialize work = available resources
2. Find process i where:
   - finish[i] == false
   - need[i] <= work
3. If such process found:
   - Simulate execution & release resources
   - work = work + allocation[i]
   - finish[i] = true
   - Repeat step 2
4. If all processes finish: SAFE STATE
5. If processes remain unfinished: UNSAFE STATE
```

7. Integration and System Flow

7.1 Complete System Flow

1. Producer threads generate processes dynamically.
2. Processes inserted into bounded buffer (semaphore protected).
3. Consumer thread fetches processes from buffer.
4. Processes added to ready queue.
5. Banker's Algorithm checks resource allocation safety.
6. Safe processes scheduled using appropriate algorithm.
7. Resources released when processes complete.

8. Limitations of Current Simulator

- **Simplified Banker's Algorithm**: Fixed number of resource types (currently 2: R1, R2).

- **Fixed Time Quantum**: Round Robin time quantum is fixed at 2 units.
- **Fixed Buffer Size**: Bounded buffer size is fixed at 10 processes.
- **Single CPU Assumption**: Simulator assumes single CPU core.
- **No I/O Operations**: Processes are CPU-bound only.

9. Conclusion

This Mini OS Simulator successfully integrates three fundamental operating system concepts: CPU scheduling, process synchronization, and deadlock prevention. The modular design allows for easy extension and modification.

Key Achievements:

- ✅ **Dynamic Scheduler Selection**: Automatically chooses optimal scheduler.
- ✅ **Proper Synchronization**: Uses semaphores and mutexes correctly.
- ✅ **Deadlock Prevention**: Banker's Algorithm ensures system safety.

References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
2. Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.
3. Dijkstra, E. W. (1965). "Cooperating Sequential Processes". *Technical Report EWD-123*.

End of Report