COMPREHENSIVE CPU SCHEDULING SYSTEM

Multi-Threaded Process Scheduling with Deadlock Prevention

Integrating CPU Scheduling • Producer-Consumer Synchronization • Banker's Algorithm

Group Names	ID's
Abid Ali	F2023266407
M Taha Jamil	F2023266379
M Saad Sohail	F2023266493
Taha Bajwa	F2023266390

Technical Report • January 2026
Operating Systems - Concurrent Computing Project

1. MODULE-WISE ARCHITECTURE

1.1 System Architecture Overview

The system employs a modular architecture following the Single Responsibility Principle (SRP), where each module handles a distinct aspect of the simulation. This design ensures maintainability, scalability, and clear separation of concerns. The architecture comprises three primary layers: **Data Structures**, **Synchronization Mechanisms**, and **Scheduling Algorithms**.

Module	File(s)	Primary Responsibility	Key Components
Data Structures	Process.h	Process & Gantt Entry definitions	Process attributes, resource vectors
Buffer Sync	BoundedBuffer.h/cpp	Thread-safe producer-consumer buffer	Semaphores, mutex, queue
Deadlock Prevention	BankersAlgorithm.h/cpp	Resource safety verification	Safety algorithm, matrices
CPU Scheduling	Scheduler.h/cpp	Process execution strategies	Priority, Round Robin
Thread Management	ProducerConsumer.h/cpp	Concurrent process generation	Producer/consumer threads
Main Controller	main.cpp	User interface & coordination	Menu system, orchestration

1.2 Core Module Descriptions

Process.h - Data Structure Layer: Defines the fundamental *Process* structure containing all attributes necessary for scheduling (PID, arrival time, burst time, priority) and resource management (resource requirements vector, allocated resources vector). Additionally defines *GanttEntry* for visualization. This header employs include guards to prevent multiple inclusion issues.

BoundedBuffer.h/cpp - Synchronization Layer: Implements a thread-safe bounded buffer using *semaphores* and *mutex locks*. Two semaphores (*empty* and *full*) track available and occupied slots, preventing busy waiting through *sem_wait()* blocking. The mutex ensures atomic access to the underlying queue data structure. This module is critical for proper producer-consumer synchronization without race conditions.

BankersAlgorithm.h/cpp - Deadlock Prevention Layer: Implements the Banker's Algorithm with resource allocation matrices (*Available*, *Max*, *Allocation*, *Need*). The *isSafe()* method performs safety checking by simulating resource allocation and verifying if a safe sequence exists. Uses a work vector to simulate resource availability as processes complete. Thread-safe through mutex protection on all resource operations.

Scheduler.h/cpp - Execution Layer: Contains two scheduling algorithms: *Priority Scheduling* (non-preemptive, selects highest priority process) and *Round Robin* (preemptive, time quantum-based). Integrates with Banker's Algorithm for resource safety checks before execution. Maintains ready queue, manages process state transitions, and generates Gantt charts for visualization. Handles edge cases like all-blocked scenarios gracefully.

ProducerConsumer.h/cpp - Thread Layer: Defines thread functions and argument structures. *producerThread()* generates random processes with varying burst times, priorities, and resource requirements. *consumerThread()* fetches processes from buffer and adds to scheduler. Implements thread-safe process ID generation using mutex locks. Uses *usleep()* to simulate realistic timing.

main.cpp - Control Layer: Provides menu-based interface with four options: simulation start, manual process addition, system state display, and exit. Orchestrates thread creation and joining. Manages global scheduler and Banker instances. Conditionally requests time quantum only when Round Robin will be used (>5 processes). Ensures proper cleanup of all allocated resources on exit.

2. SCHEDULING DECISION LOGIC

2.1 Algorithm Selection Strategy

The system employs an **adaptive scheduling strategy** based on workload characteristics. The decision criterion is the number of *ready processes at time t=0*, ensuring optimal algorithm selection for the given workload.

Condition	Algorithm	Type	Rationale
Ready $\leq$ 5	Priority Scheduling	Non-preemptive	Minimizes context switches, Optimal or light loads
Ready > 5	Round Robin	Preemptive	Ensures fairness, prevents starvation in heavy loads

2.2 Priority Scheduling (Non-Preemptive)

Selection Criteria: At each scheduling point, the process with the *lowest priority number* (highest priority) is selected from the ready queue. Tie-breaking uses arrival time (FCFS for equal priorities).

Execution Model: Once selected, a process runs to completion without preemption. This minimizes context switching overhead but may cause priority inversion if not managed carefully.

Resource Integration: Before execution, Banker's Algorithm verifies resource allocation safety. If allocation would lead to an unsafe state, the process is *blocked* and the next highest-priority safe process is selected. This prevents deadlock while respecting priority order.

Advantages: Low overhead, simple implementation, good for batch systems with varying process importance.
Disadvantages: May cause starvation of low-priority processes, no guarantee of response time.

2.3 Round Robin Scheduling (Preemptive)

Time Quantum Selection: User-defined quantum (typically 2-4 time units). Shorter quanta increase responsiveness but raise context-switch overhead; longer quanta improve throughput but reduce interactivity.

Queue Management: Processes are maintained in a FIFO ready queue. After quantum expiration, the running process moves to the queue's tail if not completed, ensuring fair CPU distribution.

Preemption Mechanism: Process execution is interrupted after *min(quantum, remaining_time)* units. This ensures no process monopolizes the CPU while allowing completion within a quantum if possible.

Resource Consideration: Banker's check occurs only at *first execution attempt* (not after preemption),

Advantages: Fair CPU allocation, bounded waiting time, good interactive performance.
Disadvantages: Context-switch overhead, quantum selection critical, poor for I/O-bound processes.

3. SEMAPHORE SYNCHRONIZATION MECHANISM

3.1 Counting Semaphores for Buffer Management

The bounded buffer employs two **counting semaphores** to solve the producer-consumer problem without busy waiting. This classic synchronization pattern ensures thread safety while maximizing concurrency.

Semaphore	Initial Value	Purpose	Wait Condition
empty	buffer_size	Tracks empty slots	Producer waits if buffer full
full	0	Tracks filled slots	Consumer waits if buffer empty

3.2 Producer Thread Synchronization

Step 1: *sem_wait(∅)* - Decrement empty counter. If empty=0 (buffer full), thread **blocks** until consumer signals availability. No CPU cycles wasted in busy-waiting.

Step 2: *pthread_mutex_lock(&mutex;)* - Enter critical section with exclusive access to buffer.

Step 3: Insert process into buffer queue (protected operation).

Step 4: *pthread_mutex_unlock(&mutex;)* - Release critical section lock.

Step 5: *sem_post(&full;)* - Increment full counter, potentially waking blocked consumer.

3.3 Consumer Thread Synchronization

Step 1: *sem_wait(&full;)* - Decrement full counter. If full=0 (buffer empty), thread **blocks** until producer signals data availability.

Step 2: *pthread_mutex_lock(&mutex;)* - Enter critical section for safe buffer access.

Step 3: Remove process from buffer queue (atomic operation).

Step 4: *pthread_mutex_unlock(&mutex;)* - Exit critical section.

Step 5: *sem_post(∅)* - Increment empty counter, potentially waking blocked producer.

3.4 Critical Synchronization Properties

Mutual Exclusion: Mutex ensures only one thread accesses buffer at any instant, preventing race conditions on shared queue data structure.

No Busy Waiting: *sem_wait()* uses kernel-level blocking, not spinlocks. Blocked threads consume zero CPU cycles, yielding to other runnable threads.

Progress Guarantee: If buffer has space, producers can insert. If buffer has data, consumers can remove. No artificial delays beyond actual resource availability.

Bounded Waiting: FIFO nature of semaphore wait queues ensures no thread waits indefinitely if others are making progress.

4. DEADLOCK PREVENTION VIA BANKER'S ALGORITHM

4.1 Banker's Algorithm Foundation

The Banker's Algorithm, developed by Dijkstra, prevents deadlock by ensuring the system never enters an *unsafe state*. A state is **safe** if there exists a sequence of all processes such that each can acquire its maximum resources, execute, and release resources without deadlock. Our implementation uses three resource types: R1, R2, R3 with total availability [10, 5, 7].

4.2 Resource Allocation Matrices

Matrix	Dimensions	Meaning	Example Entry
Available	1 × m	Currently available resources	[10, 5, 7]
Max	n × m	Maximum resource need per process	P1: [3, 2, 2]
Allocation	n × m	Currently allocated resources	P1: [1, 0, 1]
Need	n × m	Max - Allocation (remaining need)	P1: [2, 2, 1]

4.3 Safety Check Algorithm

Input: Current system state (Available, Allocation, Need matrices)
Output: Boolean (safe/unsafe) + Safe sequence if safe

Algorithm Steps:
1. Initialize Work = Available (simulated available resources)
2. Mark all processes as unfinished
3. Find process Pi where Need[i] ≤ Work AND Pi is unfinished
4. If found: Work = Work + Allocation[i], mark Pi finished, add to safe sequence
5. Repeat step 3-4 until all processes finished OR no process can proceed
6. If all processes finished: SAFE (return sequence). Else: UNSAFE (deadlock possible)

Complexity: $O(n^2 \times m)$ where n=processes, m=resource types. Acceptable for realistic n,m values.

4.4 Resource Request Handling

When process Pi requests resources Request[i]:

Step 1 - Validity Check: Verify Request[i] ≤ Need[i]. If violated, error (requesting more than declared max).

Step 2 - Availability Check: Verify Request[i] ≤ Available. If insufficient, block process (not enough resources).

Step 3 - Tentative Allocation: Simulate allocation:
• Available = Available - Request[i]
• Allocation[i] = Allocation[i] + Request[i]
• Need[i] = Need[i] - Request[i]

Step 4 - Safety Test: Run safety algorithm on simulated state.
• If SAFE: Commit allocation, process proceeds
• If UNSAFE: Rollback changes, block process, try next process

Step 5 - Blocked Process Handling: Maintain blocked list, periodically retry when resources released.

4.5 Deadlock Condition Prevention

Condition	Prevention Method in Our System
Mutual Exclusion	Cannot prevent (resources inherently non-sharable)
Hold and Wait	Prevented: Request all resources atomically, check safety before grant
No Preemption	Allowed: Resources held until process completion (Banker ensures safety)
Circular Wait	Prevented: Safety algorithm ensures no circular dependency possible

5. SYSTEM LIMITATIONS & FUTURE ENHANCEMENTS

5.1 Current System Limitations

1. **Fixed Resource Types:** System hardcoded to 3 resource types [R1, R2, R3]. Real systems have dynamic resource types.

2. **Simplified Arrival Model:** All processes arrive at t=0. Real systems have continuous, unpredictable arrivals over time.

3. **No I/O Consideration:** Assumes pure CPU-bound processes. Real processes alternate between CPU and I/O bursts.

4. **Static Priority:** Process priorities fixed at creation. Real systems often employ dynamic priority aging to prevent starvation.

5. **Single CPU:** Simulates uniprocessor system. Modern systems are multi-core with load balancing requirements.

6. **No Process Migration:** Once process enters ready queue, it cannot migrate between processors or priority levels.

7. **Banker's Conservativeness:** May block processes unnecessarily when a safe sequence exists but isn't immediately found.

8. **No Preemption in Priority:** High-priority processes may wait for low-priority ones to complete, causing priority inversion.

9. **Limited Blocked Process Recovery:** Blocked processes not automatically retried when resources become available.

10. **Absence of Real-Time Constraints:** No deadline handling or urgency-based scheduling for time-critical processes.

5.2 Proposed Future Enhancements

Enhancement	Benefit	Implementation Complexity
Multi-level Feedback Queue	Better interactive vs. batch separation	Medium
Dynamic Priority Aging	Eliminate low-priority starvation	Low
Multi-processor Support	Realistic modern system simulation	High
I/O Burst Modeling	CPU-I/O overlap, realistic utilization	Medium
Configurable Resource Types	Greater flexibility, real-world applicability	Low
Real-time Scheduling (EDF)	Deadline-aware execution	High

Process Migration	Load balancing, thermal management	High
GUI Visualization	Better educational value, debugging	Medium
Performance Metrics Export	Statistical analysis, benchmarking	Low

5.3 System Strengths Summary

Despite limitations, the system successfully demonstrates core OS concepts: **CPU scheduling algorithms**, **thread synchronization without busy-waiting**, and **deadlock prevention via Banker's Algorithm**. The modular architecture facilitates future extensions while maintaining code clarity. Thread safety is rigorously maintained through proper semaphore and mutex usage. The system handles edge cases (all-blocked scenarios) gracefully, showing robustness. Educational value is high due to clear separation of concerns and comprehensive documentation.

CONCLUSION

This project successfully integrates three fundamental operating system concepts—**CPU scheduling**, **concurrent programming**, and **deadlock prevention**—into a cohesive, functional simulator. The implementation demonstrates deep understanding of process management, thread synchronization primitives, and resource allocation strategies.

The modular architecture ensures maintainability and extensibility, with each component having well-defined responsibilities. Thread safety is achieved through proper use of semaphores and mutex locks, eliminating race conditions and busy-waiting. The Banker's Algorithm integration showcases how theoretical deadlock prevention translates into practical code.

Key achievements include: (1) *Adaptive scheduling* based on workload characteristics, (2) *Producer-consumer synchronization* without busy-waiting, (3) *Safe resource allocation* preventing deadlock, (4) *Graceful handling* of edge cases like all-blocked scenarios, and (5) *Clean separation* of concerns across 10 modular source files.

The simulator serves both as a functional tool for studying scheduling behaviors and as a demonstration of software engineering principles in systems programming. Future enhancements could address current limitations while preserving the system's educational clarity.

Metric	Value
Total Source Files	10 (.h and .cpp)
Lines of Code	~1500 (excluding comments)
Thread Types	3 (Producer × n, Consumer × 1, (Main)
Synchronization Primitives	2 Semaphores + 3 Mutexes
Scheduling Algorithms	2 (Priority, Round Robin)
Resource Types	3 (R1, R2, R3)
Memory Safety	100% (proper cleanup)

— End of Technical Report —
Operating Systems • Concurrent Computing Project