Multithreaded Producer-Consumer Application with Priority Handling and Performance Metrics

Khalid Alfahim - b00100122
Ahmad Mustafawi - b00094873

Operating Systems - CMP 310
American University of Sharjah

November 16, 2025

1 Design Decisions

1.1 Synchronization Strategy

Our implementation employs a two-level synchronization mechanism combining semaphores and mutexes to ensure thread safety and prevent race conditions.

Semaphores for State Management: We utilize two semaphores (`empty` and `full`) to manage buffer occupancy. The `empty` semaphore, initialized to the buffer size, tracks available slots and blocks producers when the buffer is full. Conversely, the `full` semaphore, initialized to zero, tracks occupied slots and blocks consumers when the buffer is empty. This approach eliminates busy-waiting, allowing threads to block efficiently on semaphore operations.

Mutex for Critical Section Protection: A mutex (`mutex`) protects the buffer during insertion and removal operations, ensuring mutual exclusion when modifying shared state (head, tail, and count indices). An additional `stats_mutex` protects shared statistics counters. This separation prevents contention between buffer operations and statistics updates.

Rationale: Semaphores provide efficient signaling for buffer fullness/emptiness, while mutexes ensure atomic buffer manipulation. This combination prevents both overflow/underflow conditions and data corruption without resorting to inefficient polling or sleep-based approaches.

1.2 Circular Buffer Implementation

The buffer uses a dynamically allocated array with head and tail indices that wrap around using modulo arithmetic: `(index + 1) % size`. A separate count field distinguishes between full and empty states, avoiding ambiguity when head equals tail.

Rationale: Circular buffers maximize memory efficiency by reusing array slots. The modulo operation handles wrapping automatically, and maintaining an explicit count simplifies full/empty detection without additional state variables.

1.3 Priority Handling (Bonus Feature)

Each buffer item contains a priority field (0=normal, 1=urgent). When inserting an urgent item, we scan backward from the tail to identify normal priority items, shift them one position toward the tail, and insert the urgent item ahead of them. This maintains FIFO order within each priority level while ensuring urgent items are always processed first.

Rationale: This approach avoids the complexity of maintaining separate queues while guaranteeing priority ordering. The shifting operation, performed within the mutex-protected

critical section, ensures atomicity. Approximately 25% of items are marked urgent via random probability checking.

1.4 Poison Pill Termination

After all producer threads complete and are joined, the main thread inserts one poison pill (special value -1) per consumer into the buffer. Upon receiving a poison pill, each consumer terminates its loop gracefully. Poison pills are marked with urgent priority to ensure immediate processing.

Rationale: This technique guarantees that consumers don't exit prematurely while producers are still active, and ensures all real items are consumed before termination. It provides clean, deterministic shutdown without polling or timeout mechanisms.

1.5 Performance Metrics (Bonus Feature)

Each item records an enqueue timestamp using `gettimeofday()` when produced. Consumers record dequeue timestamps upon removal, calculating latency as the difference. Average latency and throughput (items/second) are computed and displayed after all threads complete.

Rationale: These metrics provide quantitative insight into system performance, demonstrating how buffer size affects contention, latency, and throughput. Microsecond-precision timestamps enable meaningful measurements even for fast operations.

2 Challenges and Solutions

2.1 Challenge 1: Priority Queue in Circular Buffer

Problem: Implementing priority handling in a circular buffer is non-trivial because standard circular buffers assume strict FIFO ordering. We needed to maintain FIFO within each priority level while allowing urgent items to bypass normal items.

Solution: We developed a shifting algorithm that identifies normal priority items at the tail end and shifts them one position to make room for urgent items. This preserves FIFO within priorities while allowing urgent items to "jump ahead."

```
// Find normal items at tail
for (int i = 0; i < buffer.count; i++) {
    int idx = (buffer.head + buffer.count - 1 - i
                + buffer.size) % buffer.size;
    if (buffer.items[idx].priority == PRIORITY_NORMAL)
        items_to_shift++;
    else break;
}
// Shift items and insert urgent item
```

Listing 1: Priority Insertion Logic

2.2 Challenge 2: Race Conditions in Statistics

Problem: Multiple threads concurrently updating shared counters (`total_items_produced`, `total_items_consumed`, `latency_count`) could cause race conditions and incorrect counts.

Solution: We introduced a separate `stats_mutex` specifically for protecting statistics updates. This prevents contention with the main buffer mutex and allows statistics updates to occur independently.

2.3 Challenge 3: Thread-Safe Random Number Generation

Problem: The standard `rand()` function uses shared state and is not thread-safe, potentially causing race conditions when called from multiple producer threads.

Solution: We use `rand_r()` with per-thread seeds: `unsigned int seed = time(NULL) + producer_id`. This allows each thread to maintain its own random state without shared state or locking overhead.

2.4 Challenge 4: Memory Management

Problem: Dynamic memory allocation for thread IDs, buffers, and latency arrays requires careful management to avoid memory leaks.

Solution: We allocate thread IDs in main, pass them to thread functions, and free them immediately after use inside each thread. Buffers and arrays are allocated during initialization and freed in a dedicated cleanup function called at program termination.

2.5 Challenge 5: Poison Pill Timing

Problem: Ensuring poison pills are inserted only after all real items have been produced, and that each consumer receives exactly one poison pill.

Solution: The main thread joins all producer threads before inserting poison pills, guaranteeing no more real items will be produced. We insert exactly `num_consumers` poison pills, and consumers track them separately from real items.

3 Performance Analysis

We tested the application with varying buffer sizes to observe performance characteristics:

Table 1: Performance Metrics for Different Buffer Sizes

Configuration	Buffer Size	Avg Latency (sec)	Throughput (items/sec)
8P, 8C	2	∼0.000065	∼20,000
8P, 8C	32	∼0.000020	∼50,000

Interpretation: Small buffers (2 slots) create high contention, causing frequent blocking and increased latency. Producers often wait for space, and consumers wait for data, reducing throughput. Larger buffers (32 slots) reduce contention significantly, allowing more concurrent operations. Producers can insert multiple items before blocking, and consumers have items readily available, resulting in 2.5× higher throughput and 3× lower latency. This demonstrates the critical impact of buffer sizing on concurrent system performance.

4 Testing and Verification

The application was rigorously tested with various configurations:

- **Correctness:** Verified that total items produced equals total items consumed for all test cases
- **Priority Handling:** Confirmed through output logs that urgent items are consistently consumed before later-enqueued normal items
- **Concurrency:** Tested with up to 20 threads (10 producers, 10 consumers) without deadlocks or data corruption

- **Edge Cases:** Single thread (1P, 1C), minimal buffer (size 1), and large configurations all operate correctly
- **Memory Safety:** No memory leaks detected in validation testing

5 Individual Contributions

- [**Khalid Alfahim**]: Implemented circular buffer structure and core semaphore/mutex synchronization mechanisms (30%)
- [**Khalid Alfahim - Ahmad Mustafawi**]: Developed producer and consumer thread functions, implemented poison pill termination technique (30%)
- [**Ahmad Mustafawi**]: Implemented priority handling bonus feature with shifting algorithm (20%)
- [**Khalid Alfahim - Ahmad Mustafawi**]: Added performance metrics tracking, conducted testing, and wrote comprehensive documentation (20%)

6 Conclusion

This project successfully demonstrates multithreaded programming with proper synchronization using POSIX semaphores and mutexes. The implementation correctly handles concurrent access, provides graceful termination via the poison pill technique, and includes advanced features (priority handling and performance metrics) while maintaining code clarity. Key achievements include: correct circular buffer implementation, proper synchronization without busy-waiting, priority queue functionality, comprehensive performance metrics, and robust error handling.