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 uses semaphores and mutexes to ensure thread-safe access to the shared buffer and prevent race conditions.

Semaphores for Buffer Management: We use three semaphores for synchronization. The `mutex` semaphore (initialized to 1) provides mutual exclusion for critical sections. The `empty` semaphore (initialized to buffer size) tracks available slots and blocks producers when the buffer is full. The `full` semaphore (initialized to 0) tracks occupied slots and blocks consumers when the buffer is empty. This approach eliminates busy-waiting, as threads block efficiently on semaphore operations.

Critical Section Protection: The `mutex` semaphore protects the buffer during insertion and removal operations, ensuring only one thread modifies the buffer at a time. A separate `stats_lock` mutex protects shared statistics counters to prevent race conditions during metric updates.

Rationale: This synchronization strategy prevents overflow/underflow conditions, data corruption, and race conditions without busy-waiting or inefficient polling mechanisms.

1.2 Circular Buffer Implementation

The buffer uses a dynamically allocated array with `in` and `out` indices that wrap around using modulo arithmetic: `(index + 1) % buffer_size`. The `in` index points to where the next item will be inserted (producer), and `out` points to where the next item will be removed (consumer).

Rationale: Circular buffers maximize memory efficiency by reusing array slots. The modulo operation handles wrapping automatically, enabling continuous operation without memory reallocation.

1.3 Priority Handling (Bonus Feature)

Each buffer item contains a priority field (1=urgent, 0=normal, -1=poison pill). Priority handling is implemented on the consumer side in the `remove_item()` function. The function first counts items in the buffer, then performs a linear scan to find the highest-priority item. Once found, that item is extracted and saved. If the selected item is not at the front position, items between `out` and the selected position are shifted forward to fill the gap. This ensures urgent items are always consumed before normal items, with poison pills consumed last.

Rationale: Implementing priority on the consumer side with defensive counting and explicit item extraction ensures correctness. The algorithm guarantees that urgent items (priority = 1) are consumed first, normal items (priority = 0) second, and poison pills (priority = -1) last, preventing premature consumer termination. 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. Poison pills are assigned priority -1, ensuring they are consumed only after all real items. Upon receiving a poison pill, each consumer terminates its loop gracefully.

Rationale: This technique guarantees that consumers don't exit prematurely while producers are still active, and ensures all real items are consumed before termination. The priority-based approach prevents poison pills from being consumed early, which would cause item loss.

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 challenging because standard circular buffers follow strict FIFO ordering. We needed to maintain FIFO within each priority level while allowing urgent items to bypass normal items.

Solution: We implemented priority handling on the consumer side with a defensive linear scan. The algorithm first counts items in the buffer for safe iteration (preventing infinite loops), then scans all items to find the highest priority. After extracting the selected item, it shifts intervening items forward to maintain buffer integrity.

```
// Count items for safe iteration
int count = 0;
int temp_pos = out;
while (temp_pos != in && count < buffer_size) {
    count++;
    temp_pos = (temp_pos + 1) % buffer_size;
}

// Scan for highest priority item
for (int i = 0; i < count; i++) {
    if (buffer[current_pos].priority > best_priority) {
        best_priority = buffer[current_pos].priority;
        best_pos = current_pos;
    }
    current_pos = (current_pos + 1) % buffer_size;
}
```

Listing 1: Priority Removal Logic

2.2 Challenge 2: Premature Consumer Termination

Problem: Initial implementation assigned poison pills priority = 1 (urgent), causing them to be consumed before remaining normal items. This resulted in consumers terminating early, leaving items unconsumed.

Solution: Changed poison pill priority to -1, ensuring they are consumed only after all real items (urgent and normal) have been processed. This guarantees correct termination without item loss.

2.3 Challenge 3: Race Conditions in Statistics

Problem: Multiple threads concurrently updating shared counters (`total_produced`, `total_consumed`, `total_latency`) could cause race conditions and incorrect counts.

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

2.4 Challenge 4: 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.

2.5 Challenge 5: 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.

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)
3P, 2C	10	∼0.000150	∼24,000
8P, 8C	32	∼0.000020	∼50,000

Interpretation: Small buffers create higher contention, causing more frequent blocking. Larger buffers reduce contention significantly, allowing more concurrent operations, resulting in higher throughput and lower latency. Priority handling adds minimal overhead due to efficient linear scanning within the mutex-protected critical section.

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 (100% success rate over 20+ test runs)
- **Priority Handling:** Confirmed through output logs that urgent items are consistently consumed before later-enqueued normal items, and poison pills are consumed last
- **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 defensive scanning 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 to a shared circular buffer, provides graceful termination via the poison pill technique, and includes both bonus features (priority handling and performance metrics) while maintaining code correctness and reliability.

Key achievements include:

- Correct circular buffer implementation with `in`/`out` indexing
- Proper synchronization using semaphores (`mutex`, `empty`, `full`)
- No busy-waiting, deadlocks, or race conditions
- Priority queue functionality with defensive iteration and safe shifting
- Comprehensive performance metrics (latency and throughput tracking)
- Robust error handling and memory management
- 100% reliability: All real items consumed before poison pill termination