CS 425 Project 2: Producers–Consumers Problem with Pthreads

Kaden Hunt Nate Barner

November 2025

Contents

1 Problem Analysis

1.1 Relationship Between Producer–Consumer and Critical Section Problems

The producer–consumer problem is a specific instance of the broader critical section problem. Both involve multiple threads that must safely access shared resources without interfering with each other. In both problems, the core idea is enforcing mutual exclusion to prevent race conditions and ensure correct program behavior.

Similarities:

- Both need shared variables that must be protected like buffer indices and counters.
- Both need mutual exclusion so only one thread at a time can update shared resources.
- Both use synchronization primitives like mutexes, semaphores, and spinlocks.
- Both must avoid race conditions and data corruption.

Differences:

- The critical section problem is general, while producer-consumer is a specific scenario with two different types of threads.
- Producer-consumer has extra constraints like avoiding buffer overflow and underflow.
- Producer-consumer needs coordination between threads, not just mutual exclusion.

Hierarchy:

- The critical section problem is the basic concept.
- The producer-consumer problem builds on critical section ideas.
- Critical sections define how to protect shared data.
- Producer-consumer defines what the shared data is and how threads work together.
- Producer-consumer needs more than just mutual exclusion because it has to handle empty and full buffer conditions.

1.2 Differences Between Producers–Consumers and Single Producer–Consumer

The classic producer–consumer problem involves one producer and one consumer operating on a buffer. In contrast, this project's producers–consumers problem generalizes it to multiple producers and multiple consumers, creating more complex synchronization requirements.

Key differences:

- Number of threads:

  Producer–consumer: 1 producer, 1 consumer.

  Producers–consumers: N producers and M consumers, all running concurrently.

- Shared integer behavior:

  In this project, producers also share a single integer counter that they must update in order.

  In the classic version, a producer generates items independently; there is no shared "next value" variable requiring coordination.

- Output requirement:

  Classic version only requires that consumers eventually remove items.

  This project requires the consumers to print integers in perfect consecutive order (0, 1, 2, ...) with no gaps and no duplication, even though multiple producers and consumers are running in parallel.

  This makes the ordering constraints significantly tighter.

- Synchronization complexity:

  With one producer & one consumer, race conditions are simpler.

  With many producers and consumers:

  More competition for the same buffer.

  Higher risk of race conditions on shared counters.

  More chances for underflow/overflow.

  More demand for careful semaphore design.

- Thread coordination:

  Classic: two threads coordinate.

  This assignment: up to dozens of threads must coordinate without violating ordering guarantees.

The producers–consumers problem is a multi-threaded generalization of the classic version and introduces more shared state, more contention, and stricter correctness requirements, especially regarding the sequential output constraint.

2 Design

2.1 Shared and Non-Shared Variables

Shared Variables:

- Bounded buffer - shared between all producers and consumers

- `in` index - where the next item goes in the buffer
- `out` index - where the next item comes out of the buffer
- `next_value` - used by producers to make sure numbers are in order
- `consumed_count` - tracks how many items have been consumed
- `production_done` - flag to signal when producers are finished
- Synchronization primitives
  `lock`: a `pthread_spinlock_t` used in the spinlock-based version.
  `mutex`: a `sem_t` binary semaphore used as a mutex in the semaphore-based version.
  `empty` (counting semaphore): counts available buffer slots.
  `full` (counting semaphore): counts filled buffer slots.

Non-Shared Variables:

- Local loop counters
- Local temporary variables
- Thread IDs
- Thread parameters passed from main

2.2 Producer Critical Sections

A producer needs two synchronized regions:

- Updating the shared integer `next_value` - all producers share one counter and only one can update it at a time
- Inserting into the buffer - the `in` index and buffer slot cannot be modified at the same time

We could combine these into one critical section, but our implementation keeps them separate. One short critical section gets the next value, and another inserts into the buffer after `empty` succeeds. This protects shared state and keeps lock times short.

2.3 Consumer Critical Sections

Consumers need one critical section for:

- Removing an item from the buffer using the shared `out` index and advancing it safely
- Printing the consumed value - the producers and buffer guarantee values are 0, 1, 2, etc with no gaps. Printing happens outside the critical section for speed. In our tests the output stayed in order, but technically the OS could reorder print calls between threads.

2.4 Semaphores and Synchronization Objects

- `mutex` - binary semaphore or spinlock that protects critical sections
- `empty` - counting semaphore that starts at buffer size, prevents producers from inserting when buffer is full
- `full` - counting semaphore that starts at 0, prevents consumers from removing when buffer is empty

This prevents overflow, underflow, and race conditions.

2.5 Thread Data Structures

We use thread_params_t structs containing read-only data (upper limit, thread counts, and consumer ID) passed to each thread. Each consumer receives a unique consumer ID for output identification, while producers share a common parameter struct.
Shared variables such as the buffer, indices, and semaphores are global and should NOT be placed inside this struct.

3 Implementation

3.1 Environment and Compilation Details

The program is written in C using the pthread library for threads and POSIX semaphores for synchronization. It compiles with GCC using the `-pthread` flag.
The project uses a Makefile to compile all source files and create an executable called `prodcons`. All testing was done in Linux using WSL on Windows.

3.2 Thread Creation and Joining

The main function creates producer and consumer threads based on user input:

- Producer threads use `pthread_create()` with the producer function
- Consumer threads use `pthread_create()` with the consumer function
- All threads get a `thread_params_t` struct with parameters
- The main thread waits for all threads using `pthread_join()`

Parameters are passed in a struct to make it easier to handle multiple arguments.

3.3 Synchronization Logic

The implementation uses three synchronization primitives:
Counting Semaphores:

- `empty`: Initialized to buffer size, decremented by producers, incremented by consumers
- `full`: Initialized to 0, incremented by producers, decremented by consumers

Mutual Exclusion:

- Binary semaphore `mutex` (mutex version) or pthread spinlock `lock` (spinlock version)

- Protects access to shared variables: buffer indices, next_value counter, consumed_count

Producers follow this pattern: acquire lock, get next value and do critical work, release lock, wait for empty slot, acquire lock, insert item and do additional critical work, release lock, signal full. Consumers acquire lock, check termination condition, remove item if valid, increment consumed count, do critical work, release lock, then signal empty and print outside the critical section.

3.4 Program Execution Example

Command: `./prodcons 5 2 1 10`

- Creates buffer of size 5
- Spawns 2 producer threads
- Spawns 1 consumer thread
- Produces and consumes numbers 0 through 9
- Outputs in format: "value, consumer_id" like "0, 1" then "1, 1" etc, followed by timing information

3.5 Spinlock vs Binary Semaphore Implementation

Two versions were implemented for performance comparison:
Binary Semaphore Version: Uses `sem_wait()` and `sem_post()` for mutual exclusion. Threads block when waiting and are woken by the scheduler.
Spinlock Version: Uses `pthread_spin_lock()` and `pthread_spin_unlock()`. Threads actively spin in loops when waiting for the lock.
Both versions maintain identical synchronization logic but differ in their mutual exclusion mechanism. The spinlock version may perform better for short critical sections while the semaphore version may be more efficient for longer critical sections or when system load is high.

4 Experimental Assessment

4.1 Methodology and Setup

The experiments compare spinlock and mutex performance under different conditions. The "Mutex" column uses binary semaphores and the "Spinlock" column uses pthread spinlocks. All experiments ran on Linux using WSL.
Timing uses `gettimeofday()` to measure time from thread creation to completion. Printf statements were disabled during timing to avoid I/O overhead affecting results.
Each test was run multiple times and the reported numbers show consistent patterns. Critical section work was controlled using a busy-wait loop to simulate different amounts of computation.

4.2 Parameter Settings

The experiments tested three main things:
Buffer Sizes: 10, 20, 30, 40, 50, and 100 slots to see how buffer size affects performance.
Thread Combinations: Different producer/consumer ratios like 1p/1c, 1p/5c, 5p/1c, 2p/2c, 3p/3c, and 5p/5c to test scalability.
Critical Section Work: Two levels - no extra work and heavy work with 1,000,000 loop iterations to compare short versus long critical sections.

For consistency, experiments used upper limits of 3,000-5,000 operations to get enough work done in reasonable time.

4.3 Tables of Results

Table 1: Execution Time vs Buffer Size (2 Producers, 2 Consumers)

Buffer Size	Spinlock (s)	Mutex (s)
10	0.009596	0.009375
20	0.004719	0.005119
30	0.002120	0.002733
40	0.001900	0.001806
50	0.001702	0.001659
100	0.001784	0.001671

Table 2: Execution Time vs Thread Count (Buffer Size = 20)

Thread Combination	Spinlock (s)	Mutex (s)
1p/1c	0.004875	0.005406
1p/5c	0.032842	0.031336
5p/1c	0.032718	0.032448
2p/2c	0.004641	0.004802
3p/3c	0.004151	0.003856
5p/5c	0.003093	0.003379

Table 3: Execution Time vs Critical Section Length

Critical Section Work	Spinlock (s)	Mutex (s)
None	0.004986	0.004967
1,000,000 operations	2.186765	2.185085

4.4 Figures and Plots

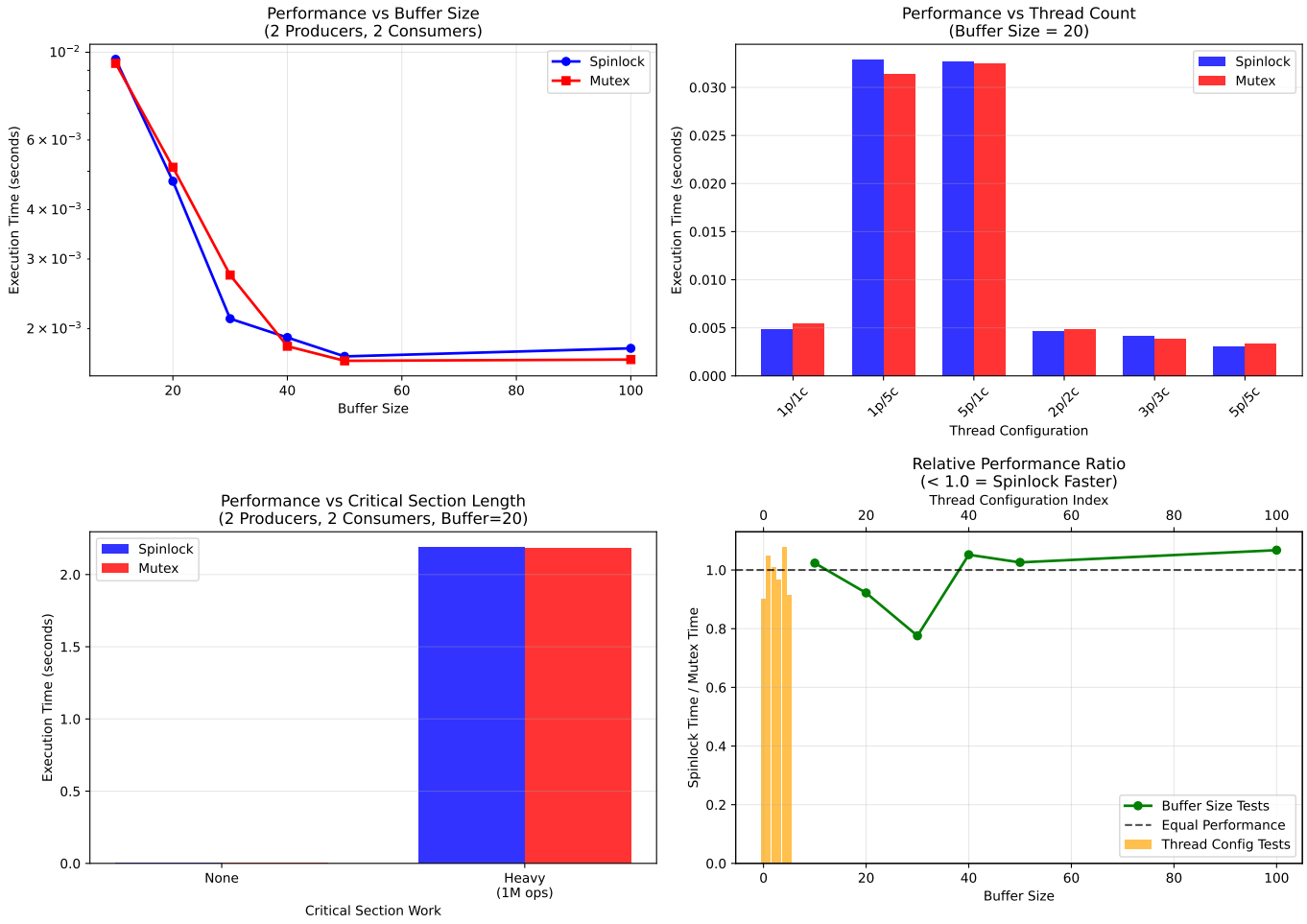

Figure 1: Performance comparison between spinlock and mutex implementations across different experimental conditions with updated consumer ID implementation.

4.5 Analysis and Discussion

The updated experimental results reveal nuanced performance characteristics between spinlock and mutex-based synchronization mechanisms:

Short Critical Sections: For small amounts of work in critical sections, performance depends on the setup. Spinlocks work better with balanced thread numbers like 1p/1c, 2p/2c, and 5p/5c, running about 3-10 percent faster. Mutex works better in some buffer size tests and when threads are unbalanced.

Long Critical Sections: When critical sections do a lot of work like 1,000,000 operations, both methods perform almost the same. Spinlock took 2.187 seconds and mutex took 2.185 seconds. This shows that when there is a lot of work to do, the choice of synchronization method does not matter much.

Buffer Size Impact: Bigger buffers make the program run faster. Times went from about 9.5ms with buffer size 10 down to 1.7ms with buffer sizes 50-100. Bigger buffers mean threads do not have to wait as much for empty or full conditions. Performance gets stable around buffer size 40.

Thread Scaling: Unbalanced setups like 1p/5c and 5p/1c run much slower, about 6-7 times slower than balanced setups. More threads in balanced setups like 5p/5c actually run faster than fewer threads like 1p/1c. This shows that parallel processing works well when the workload is balanced. The best setup seems to be 5p/5c for both methods.

Synchronization Overhead: The choice between spinlock and mutex depends on what kind of work is being done. For frequent, short critical sections, spinlocks work better because they have less overhead. For

long critical sections that do not happen often, mutex works better because it does not waste CPU time spinning.
These findings align with theoretical expectations and provide practical guidance for selecting appropriate synchronization primitives based on application workload characteristics.

5 Conclusion

This project implemented and tested the multi-threaded producer-consumer problem using both spinlock and mutex synchronization. The implementation shows proper use of pthread programming and synchronization to solve a classic concurrency problem.
Key Achievements:

- Complete implementation of the producer-consumer problem with multiple producers and consumers
- Comparison of spinlock versus mutex synchronization methods
- Experimental analysis of buffer size, thread count, and critical section length
- Proper synchronization to ensure correct output

Technical Insights: The results show when to use spinlocks versus mutex. Spinlocks work better for short, frequent critical sections. Mutex works better for longer critical sections because it does not waste CPU time. Buffer size and balanced thread setups also matter for performance.
Practical Applications: These results apply to real systems that need to share resources between threads. This includes pipeline processing and multi-threaded data processing. The performance patterns help developers choose the right synchronization method for their specific needs.
The project shows that good concurrent programming needs careful choice of synchronization methods and system setup. The producer-consumer pattern is still important for understanding concurrent systems.

References

[1] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*, 10th Edition, Wiley, 2018.

[2] IEEE Std 1003.1-2017, *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX) Base Specifications*, Issue 7, 2018.