Assignment 3: Solving Producer Consumer Problem using Semaphores and Locks

Uppu Eshwar
Roll No.: Ch21btech11034

Contents

1 Introduction

This assignment implements and evaluates solutions for the **bounded buffer** variant, where producer threads add items to a fixed-size shared buffer, and consumer threads remove them. The objective is to implement this using two distinct C++ synchronization approaches and compare their performance:

1. **Semaphores:** Employing standard counting and binary semaphores for efficient coordination and mutual exclusion.
2. **Locks (Mutexes):** Relying solely on a mutex and implementing condition checking via a busy-wait polling mechanism.

Performance is evaluated by measuring the average time threads spend within their critical sections (CS) while accessing the buffer. Experiments vary the relative speeds of producers and consumers (Delay Ratio) and the ratio of producer to consumer threads (Thread Ratio). This report covers the design, implementation differences, experimental methodology, results based on the new parameters, and an analysis of the findings.

2 Design of the Problem

The system manages concurrent access to a shared circular buffer of a predefined capacity.

2.1 Shared Buffer

A std::vector< *int* > serves as the circular buffer. in_index points to the next insertion slot for producers, and out_index points to the next removal slot for consumers. Indices wrap around using the modulo operator (% capacity).

2.2 Producer Threads

np producer threads each execute a loop cntp times (where cntp is now fixed at 100 in both experiments) :

1. Wait for an empty slot.
2. Enter Critical Section (acquire lock/mutex).
3. Place item at in_index, update in_index.
4. Exit Critical Section (release lock/mutex).
5. Signal item availability.
6. Simulate external work via an exponential delay (mean mu_p).

2.3 Consumer Threads

nc consumer threads each execute a loop cntc times (where cntc is fixed at 100 in Exp 1 and calculated in Exp 2 based on np and cntp):

1. Wait for an available item.
2. Enter Critical Section (acquire lock/mutex).
3. Get item from out_index, update out_index.

4. Exit Critical Section (release lock/mutex).
5. Signal slot availability.
6. Simulate external work via an exponential delay (mean mu_c).

2.4 Input Parameters

Read from inp-params.txt: capacity, np, nc, cntp, cntc, mu_p, mu_c. The values for cntp, cntc, mu_p, mu_c now follow the updated specifications for each experiment.

3 Synchronization Mechanisms and Implementation

Two C++ implementations (prod_cons-sems-ch21btech11034.cpp, prod_cons-locks-ch21btech11034.cpp) using pthreads were created.

3.1 Semaphore Implementation (prod_cons-sems.cpp)

This standard solution uses three semaphores (semaphore.h):

- sem_empty: Initialized to capacity, controls producer access.
- sem_full: Initialized to 0, controls consumer access.
- sem_mutex: Initialized to 1, ensures mutual exclusion for buffer access.

Threads efficiently block using sem_wait when necessary and signal completion using sem_post.

3.2 Lock (Mutex) Implementation (prod_cons-locks.cpp)

This version uses only a single buffer_lock. An integer count_items tracks buffer occupancy. Crucially, waiting for buffer conditions (not full/not empty) is handled via busy-wait polling with explicit lock release and sleep:

- A thread acquires buffer_lock.
- It checks the condition (e.g., count_items == capacity).
- Problem with Simple Spinlock: A simple loop like while(condition) while holding the lock would cause deadlock. The waiting thread holds the only lock, preventing other threads (which could change the condition) from ever acquiring it.
- Implemented Solution: If the condition blocks progress, the thread: 1. Releases the lock (pthread_mutex_unlock). 2. Sleeps briefly (this_thread::sleep_for(chrono::milliseconds(1))) 3. Re-acquires the lock (pthread_mutex_lock) to re-check the condition.
- This unlock-sleep-lock cycle prevents deadlock but introduces significant overhead and potential inefficiency compared to semaphores or condition variables.

Busy-Wait Logic Example (Producer):

```
pthread_mutex_lock(&buffer_lock);
while (count_items == capacity) { // Check if full
    pthread_mutex_unlock(&buffer_lock); // Release lock!
    this_thread::sleep_for(chrono::milliseconds(1)); // Sleep briefly
    pthread_mutex_lock(&buffer_lock); // Re-acquire lock
}
pthread_mutex_unlock(&buffer_lock);
```

3.3 Timing and Logging

Both implementations use std::chrono::steady_clock for timing.

- CS entry/exit times relative to a program start time are recorded in nanoseconds.
- Logs are buffered per-thread and then combined, sorted chronologically based on the nanosecond timestamp appended to each line.

4 Experimental Setup and Results

4.1 Experiment Automation and Analysis

The Python script (experiments.py) automates the process:

- Generates inp-params.txt based on the parameters for each experiment run.
- Executes the compiled C++ programs.
- Parses PROD_CS: and CONS_CS: lines from the output files to extract nanosecond CS entry/exit times.
- Calculates CS durations and converts them to milliseconds.
- Averages results over NUM_TRIALS = 3 runs per configuration.
- Generates comparison plots using matplotlib with a logarithmic Y-axis for better visualization of time differences.

4.2 Experiment 1: Delay Ratio

Fixed np=5, nc=5, cntp=100, cntc=100, and mu_p = 10.0 ms. The ratio r = mu_p / mu_c was varied over $\{10, 5, 1, 0.5, 0.1\}$, which required calculating mu_c = mu_p / r for each run.

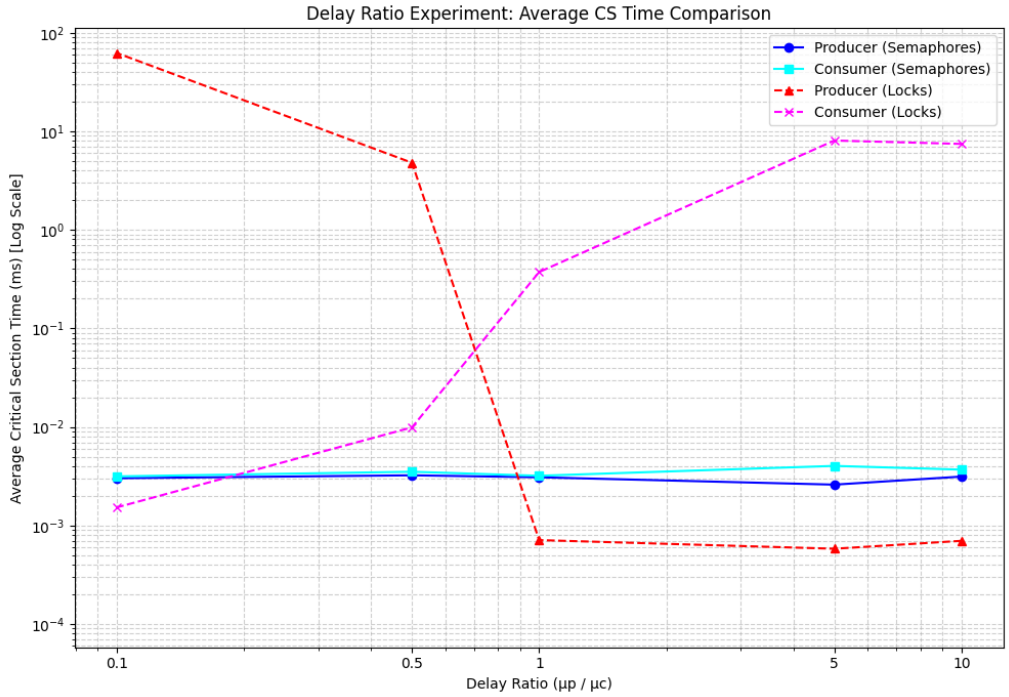

Figure 1: Experiment 1: Avg CS Time (ms, Log Scale) vs. Delay Ratio (μ_p/μ_c).

Observations (New Plot):

- Semaphore CS times (blue/cyan) remain extremely low (around 0.002-0.005 ms) and stable across all ratios.
- Lock Consumer CS times (magenta) increase as the ratio increases (consumers become relatively slower, mu_c gets smaller). They spend more time polling on the empty buffer, leading to higher average CS times (reaching 10ms at ratio 10).
- Lock Producer CS times (red) exhibit a surprising sharp peak at low ratios (0.1 and 0.5), reaching nearly 100ms at ratio 0.1, before dropping drastically at higher ratios (below 0.001ms for ratio >= 1).
- Explanation for Producer Peak: At low ratios (0.1, 0.5), consumers are extremely fast (mu_c is large: 100ms, 20ms). They likely empty the buffer quickly and then enter their fast polling loop (unlock-sleep(1ms)-lock). The high number of consumers (5) rapidly polling creates intense contention for the single buffer_lock. The producer, finishing its 1ms work, struggles to acquire the lock against these numerous, frequent attempts by consumers, leading to producer starvation and inflating its average CS time (as it includes the time spent fighting for the lock in the polling loop entrance). Once the ratio increases (consumers slow down), this contention reduces dramatically, and the producer lock time plummets.

4.3 Experiment 2: Thread Ratio

Fixed mu_p = mu_c = 10.0 ms and cntp = 100. The ratio np / nc was varied using configurations corresponding to ratios $\{10, 5, 1(5, 5), 0.5, 0.1\}$. For each configuration, cntc was calculated as (np * cntp) / nc to ensure all produced items were consumed. This setup means the total number of items processed (total work) varies significantly between configurations (from 100 items for ratio 0.1/0.5 up to 1000 items for ratio 10).

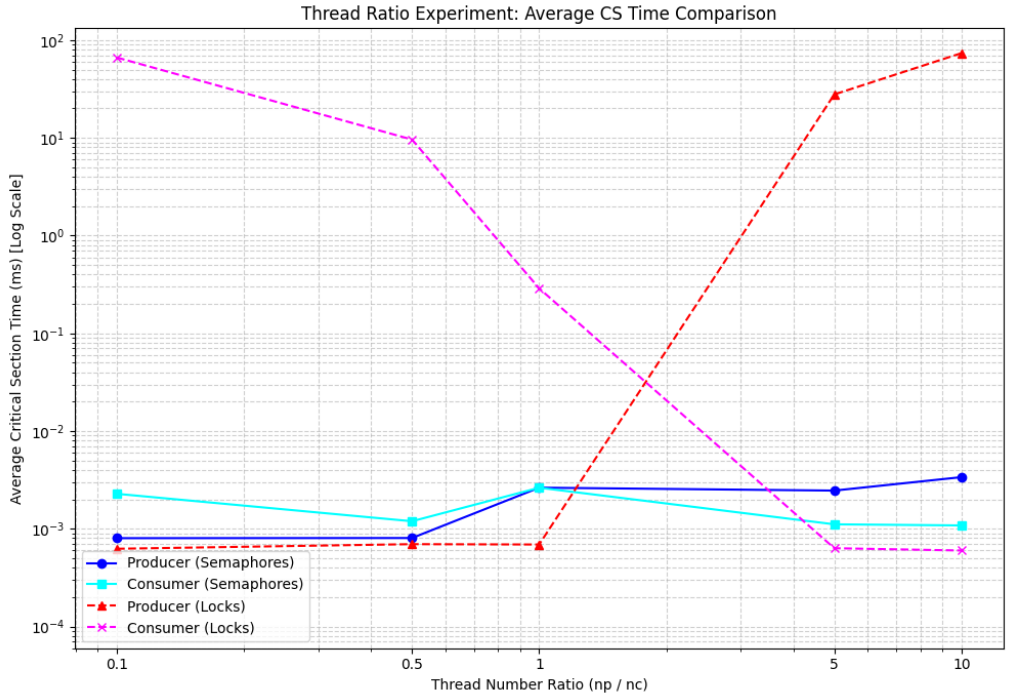

Figure 2: Experiment 2: Avg CS Time (ms, Log Scale) vs. Thread Ratio (np/nc).

Observations (New Plot):

- Semaphore CS times remain relatively low but show a slight increase compared to Exp 1, likely due to the higher workload (cntp=100). Consumer semaphore times increase slightly at low ratios (many consumers).
- Lock Consumer CS times (magenta) are extremely high (peak 70ms) at low ratios (0.1, 0.5) where many consumers (nc=10, 2) compete and poll frequently on an often-empty buffer (only 1 producer). The time drops significantly as the ratio increases.
- Lock Producer CS times (red) are very low at low ratios but increase dramatically (peak 60ms) at high ratios (5, 10) where many producers (np=5, 10) create high contention for the single buffer_lock when trying to add items.
- The varying total workload might also influence absolute times, but the trends strongly correlate with contention caused by the producer/consumer imbalance interacting with the lock-based polling mechanism.

5 Analysis

- **Semaphore Efficiency:** Semaphores consistently deliver low and stable average critical section times across all tested conditions. Their mechanism of efficiently blocking threads via sem_wait until a resource is actually available avoids the overhead seen in the lock-based approach.
- **Lock Polling Overhead:** The lock implementation's performance is dominated by the inefficiency of its busy-wait polling loop (unlock-sleep-lock). The measured CS time for locks includes this polling time, leading to dramatically inflated values whenever threads frequently encounter a blocking condition (buffer full for producers, buffer empty for consumers).

6 Conclusion

The semaphore-based implementation is the superior solution for the bounded buffer problem in terms of performance and scalability. It handles waiting efficiently and minimizes contention. The lock-only implementation, constrained to use busy-wait polling, suffers significant performance degradation due to polling overhead and lock contention, particularly when producer/-consumer speeds or numbers are imbalanced.

7 Code and Execution Structure

The project comprises:

- **C++ Source Files:**
  - prod_cons-sems-ch21btech11034.cpp: Semaphore solution.
  - prod_cons-locks-ch21btech11034.cpp: Lock/busy-wait solution
- **Python Automation Script:** experiments.py
- **Input File:** inp-params.txt (Generated by Python script).
- **Output Log Files:** output_sems.txt, output_locks.txt

Compilation requires g++ with -pthread (and for semaphore -lrt).