REPORT

OPERATING SYSTEMS - 2
PROGRAMMING ASSIGNMENT -3

Harsha Vardhan Pulavarthi
CO23BTECH11018

Introduction

As per the given assignment, we need to solve the Producer-Consumer problem using C++/ We need to use Semaphores and Locks to solve it and compare them both.

Code Design:

1. Using Locks:

This code implements the classic producer-consumer pattern in C++ using multithreading concepts. The design centers around a Buffer class that maintains a queue with a fixed capacity, allowing producers to add items and consumers to remove them. The Buffer keeps track of its state with methods like isFull() and isEmpty() to control the flow of data between threads.

Thread synchronization is handled through mutexes and condition variables, preventing race conditions when accessing the shared buffer. Producer threads lock the buffer, check if it's full, wait if necessary, add an item, and then notify waiting consumers. Similarly, consumer threads lock the buffer, check if it's empty, wait if necessary, remove an item, and notify waiting producers. This coordination ensures thread safety while maximizing throughput.

The program allows configuration through an input file that specifies parameters like buffer capacity, number of producer/consumer threads, items to process, and timing parameters. Each thread uses exponentially distributed random delays between operations to simulate realistic workloads, making the simulation more accurate to real-world scenarios where processing times vary.

Performance measurement is built into the system, with the code tracking the total time spent by producer and consumer threads. Each operation is logged to an output file with timestamps, buffer position, and thread identifiers, allowing for detailed analysis of the system's behavior. After all threads complete their work, the program calculates and reports average thread execution times, providing insights into the efficiency of the producer-consumer implementation.

2. Using Semaphores:

This code implements the classic producer-consumer pattern in C++ using semaphores instead of condition variables. The core structure still revolves around a Buffer class with a fixed-capacity queue, but the synchronization mechanism has changed. The Buffer provides the same interface with methods for checking if it's full or empty, inserting and removing items, and tracking its size, making it a clean abstraction for the shared resource.

Thread coordination is now handled through three semaphores: 'empty' counts available slots in the buffer, 'full' counts items ready for consumption, and 'buffer_mutex' ensures mutual exclusion when accessing the buffer. Producer threads acquire an empty slot, lock the buffer, insert an item, release the lock, and signal that a new item is available. Consumer threads wait for an available item, lock the buffer, remove the item, release the lock, and signal that a slot is now empty. This semaphore-based approach provides a more direct mapping to the producer-consumer problem than the previous condition variable implementation.

The program maintains the same configurable design through an input file that specifies parameters like buffer capacity, thread counts, and timing variables. However, it adds a safety check to ensure the buffer capacity doesn't exceed the maximum semaphore value

of 10,000. Each operation still uses exponentially distributed random delays to simulate realistic processing times, and logging remains comprehensive with timestamps, buffer positions, and thread identifiers being recorded for each action.

Performance measurement is preserved with slight modifications. A dedicated 'timeMutex' now protects the accumulation of timing statistics, ensuring thread-safe updates to the total time variables. The program outputs to a different file named "output-sems.txt" to distinguish it from the condition variable implementation. After all threads complete their tasks, the program calculates and reports the same average execution times, allowing for direct comparison between the two synchronization approaches to determine which provides better performance characteristics.

Analysis of Output:

For every test case, we run it 5 times and then take an average of it.

#Experiment 1:

Table:

First Experiment				
Capacity = 1000	np =10	nc = 10	Assumed	
cntp = 100	cntc = 100	mu_p = 10ms	Given	
LOCKS				
Ratio = mu_p/mu_c	mu_c	Average Producer Thread	Average Consumer Thread Time	
10	1ms	0.011321	0.011717	
5	2ms	0.010935	0.011433	
1	10ms	0.010958	0.011464	
0.5	20ms	0.011072	0.021403	
0.1	100ms	0.011244	0.103824	
SEMAPHORES				
Ratio = mu_p/mu_c	mu_c	Average Producer Thread	Average Consumer Thread Time	
10	1ms	0.010974	0.011076	
5	2ms	0.0113	0.011437	
1	10ms	0.011171	0.011501	
0.5	20ms	0.010916	0.021768	
0.1	100ms	0.011387	0.107225	

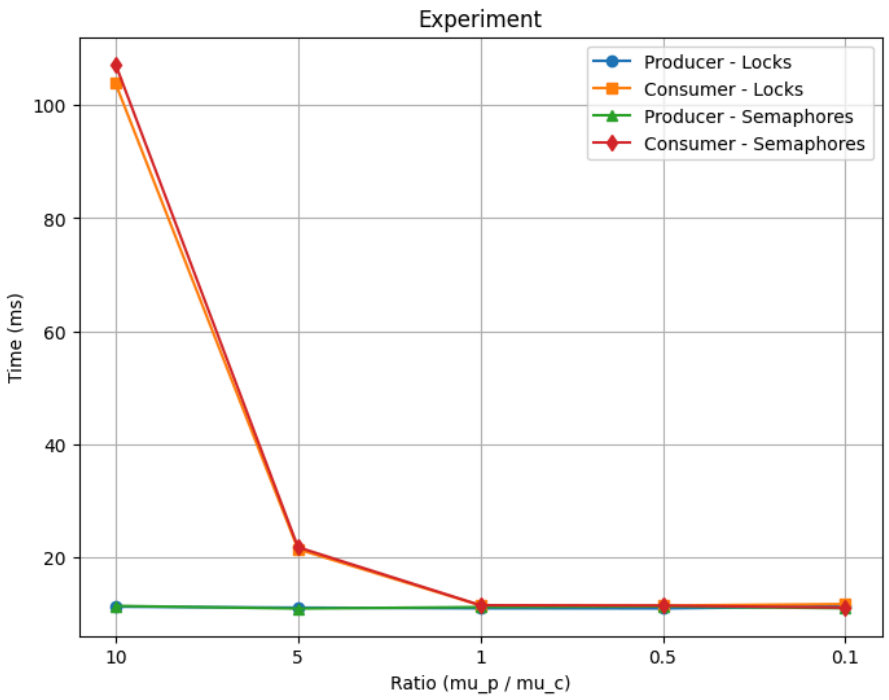
ANALYSIS:

- The experiment primarily compares the performance of producer and consumer threads under different synchronization mechanisms—locks and semaphores—while varying the ratio μp/μc. The results show that **producer thread times remain relatively stable across all ratios**, indicating that production speed is largely unaffected by changes in consumer processing time. However, **consumer thread times increase significantly** as the ratio decreases (i.e., as μc increases). This effect is more pronounced for lower μp/μc values, where consumers experience substantial delays, especially under locks, where the consumer time jumps from **0.011717 s** (μp/μc=10) to **0.103824 s** (μp/μc=0.1). A similar trend is observed with semaphores, where consumer times increase from **0.011076 s** to **0.107225 s**.

- When comparing locks and semaphores, producer thread times are nearly identical, with slight variations in favor of semaphores at certain points. However, consumer thread times under semaphores are marginally higher than those under locks for lower μp/μc values, suggesting that semaphores introduce some additional synchronization overhead. The **biggest performance gap is seen in consumer times**, where the increasing consumption time leads to significant queuing delays, particularly in cases where the consumer processing time is much higher than the producer's. This suggests that **the consumer's ability to process items efficiently is the limiting factor in system performance**, rather than the synchronization mechanism itself.

Conclusion: The experiment highlights that **consumer thread performance is highly sensitive to processing time differences**, while producer thread performance remains relatively stable. Both locks and semaphores exhibit similar trends, with semaphores offering slight improvements in producer efficiency but slightly higher consumer thread times under heavy load. The key takeaway is that **balancing producer and consumer processing speeds is critical** to ensuring optimal performance, as consumer delays become the primary bottleneck when their processing time increases significantly.

Graph:



#Experiment 2:

Table:

Second Experiment						
Capacity = 5000			Assumed			
cntp = 100	mu_p = 10ms	mu_c = 10ms	Given			
LOCKS						
Ratio = cntp/cntc	cntc	np	nc	Average Producer Thread time	Average Consumer Thread Time	
10	10	50	500	0.011043	0.092567	
5	20	50	250	0.011042	0.050458	
1	100	50	50	0.011207	0.011445	
0.5	200	50	25	0.011038	0.011135	
0.1	1000	50	5	0.010949	0.011141	
SEMAPHORES						
Ratio = cntp/cntc	cntc	np	nc	Average Producer Thread time	Average Consumer Thread Time	
10	10	50	500	0.01141	0.091569	
5	20	50	250	0.011035	0.048207	
1	100	50	50	0.011228	0.011455	
0.5	200	50	25	0.011228	0.011107	
0.1	1000	50	5	0.011056	0.011336	

ANALYSIS:

In this second experiment, the producer and consumer thread performance is analyzed with different producer-to-consumer ratios while keeping the buffer capacity at **5000**. The producer thread times remain nearly constant, similar to the previous experiment, ranging between **0.0109 s and 0.0112 s**, indicating that the producer's performance is stable irrespective of consumer count or synchronization method. However, the **consumer thread times show significant variations depending on the producer-to-consumer ratio**. When **cntp/cntc = 10**, consumer thread times are the highest (**0.092567 s for locks and 0.091569 s for semaphores**) but decrease as the ratio moves toward 1. When the ratio reaches **1 or lower**, consumer times stabilize around **0.0111 s**, indicating a balanced workload between producers and consumers.

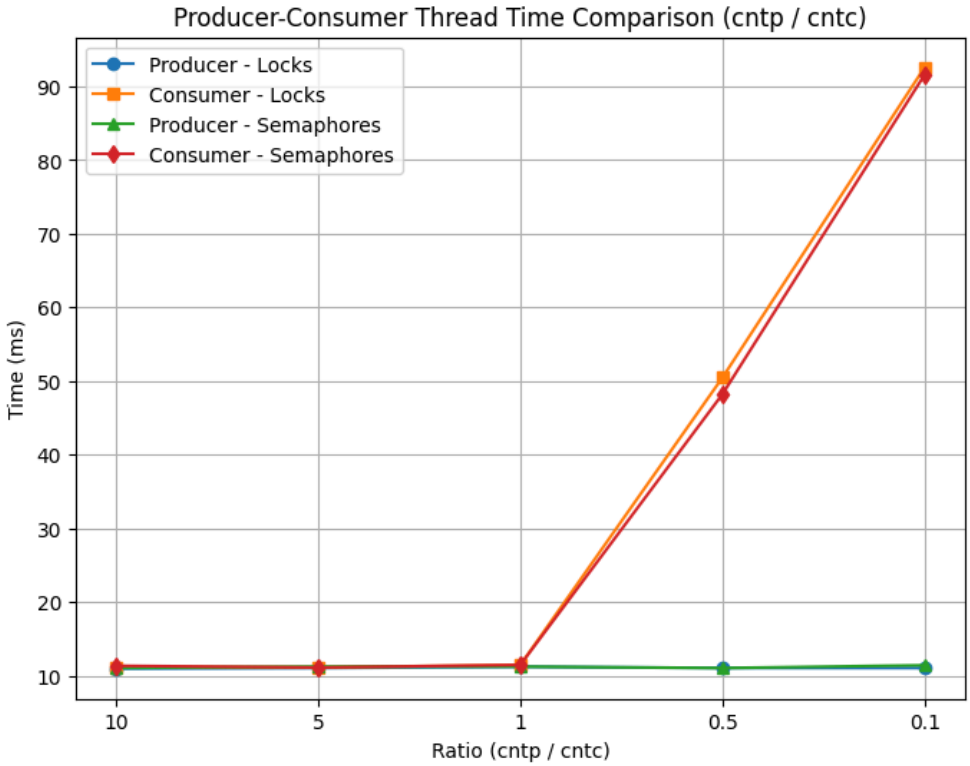
Comparing **locks and semaphores**, both methods follow the same trend, with only **slight variations** in performance. The semaphores generally produce **slightly lower consumer thread times** in most cases, indicating better efficiency under high producer-to-consumer ratios. The effect of increasing consumer threads is evident, as higher consumer counts lead to **reduced queuing delays**, allowing consumers to process items faster. At **cntp/cntc = 0.1 (where consumer count is**

highest), the consumer times for both locks and semaphores stabilize around **0.0111–0.0113 s**, proving that an increased number of consumers can effectively handle production loads.

Conclusion:

The experiment reinforces the finding that **consumer thread performance is highly sensitive to the producer-to-consumer ratio**. When **producers significantly outnumber consumers (high cntp/cntc ratio), consumer processing times rise sharply**, leading to bottlenecks. In contrast, when the ratio is balanced or tilted in favor of consumers, performance stabilizes. **Locks and semaphores perform similarly overall, with semaphores offering a slight advantage in reducing consumer thread delays.** This suggests that **optimizing consumer count relative to producer count is the key to maintaining efficient system performance** rather than the choice of synchronization method alone.

Graphs:



Overall Analysis:

The experiments highlight that **consumer performance is the key bottleneck in a producer-consumer system, not the producer efficiency**. When **too few consumers are present**, they struggle to keep up with production, causing **higher execution times**. On the other hand, **when the consumer count increases**, they effectively process the backlog, stabilizing the system. The choice of synchronization method plays a role, with **semaphores offering slight advantages over locks in reducing delays**, but the **producer-to-consumer ratio is the most critical factor in optimizing performance**. To achieve optimal system efficiency, **a balanced producer-to-consumer ratio should be maintained, ensuring neither excessive queuing nor underutilized resources.**

Note:

I have written similar codes for both Codes for ease of checking.

I have taken an average of 5 values while running the codes and plotting the graphs

Thank You!