

Multi-Threaded Producer–Consumer Problem Using Monitors and Semaphores

Operating Systems Term Project Proposal, Fall 2025

Course: COSC519

Group Members: Mayeesha Humaira & Katelyn Leedy

Instructor: Dr. Ziyang Tang

Date: 12/06/2025

Work Distribution:

1. Mayeesha Humaira implemented the Semaphore based Implementation (50%)
2. Katelyn Leedy implemented the Monitor based Implementation (50%)

I. Introduction

Process synchronization is one of the most fundamental topics in operating systems. As modern systems rely on parallel processing, it becomes essential to manage how threads and processes share data and coordinate their execution. Without proper synchronization, concurrent threads may produce inconsistent results, corrupt shared memory, or cause performance bottlenecks.

The Producer Consumer problem is a classic example that demonstrates these synchronization challenges. It models a shared resource, a bounded buffer, where producers insert data items, and consumers remove them. The challenge lies in ensuring that producers do not overwrite existing data when the buffer is full, and consumers do not read from an empty buffer, while also preventing deadlocks and starvation.

Our project extends this classic example by implementing a multi-threaded producer consumer system that uses two synchronization strategies. The first approach applies low-level primitives (semaphores and mutexes), providing explicit control of concurrency. The second approach introduces a monitor-based design that abstracts synchronization through condition variables. By implementing and comparing both methods, we aim to analyze their effectiveness in terms of correctness, efficiency, and code maintainability.

This project directly supports the course goals of understanding process management and inter-thread communication and provides practical experience with thread libraries and synchronization constructs that mirror real-world operating system mechanisms.

II. Objectives

Our main objectives are:

1. Design and implement a multi-threaded producer–consumer system using C++ with POSIX threads.
2. Develop two synchronization models:

- Model A: Semaphore-based solution using counting semaphores (empty, full) and a mutex for mutual exclusion.
 - Model B: Monitor-based solution using a single mutex and condition variables (notFull, notEmpty).
3. Compare and evaluate the two implementations for correctness, performance, CPU efficiency, fairness, and starvation.
 4. Demonstrate understanding of synchronization concepts such as process coordination, mutual exclusion, and deadlock avoidance in real operating-system scenarios.

III. Design and Implementation Approach

We will begin by designing a shared buffer structure, thread logic, and synchronization routines. A test harness will allow configurable parameters such as buffer size, number of producers and consumers, and workload intensity.

Phase 1 – Semaphore Implementation (Mayeesha)

The first synchronization method implemented in this project was the classic semaphore-based producer and consumer system. In this highly concurrent approach, the system manages thread interaction using three distinct primitives: a single mutex for mutual exclusion and two counting semaphores, empty and full, for flow control. All shared buffer states, including the circular array of storing item identifiers and its index pointers, is protected by the mutex to prevent concurrent modification. The empty semaphore, initialized to the buffer's maximum capacity, tracks available slots for producers, while the full semaphore, initialized to zero, counts the number of items available for consumers.

The operation relies on a strictly defined protocol for each thread. When a producer attempts to insert an item, it first executes a `sem_wait(&empty)`, which blocks the thread if the buffer is full, preventing overflow. Once the semaphore signals available space, the producer acquires the mutex to enter the critical section, safely inserts the new item, and then releases the lock. Finally, it executes `sem_post(&full)` to signal that a new item is ready for consumption. Consumers follow the inverse, blocking on `sem_wait(&full)` if the buffer is empty. After the full semaphore grants access, the consumer acquires the mutex to safely remove the next available item and then signals available space by executing `sem_post(&empty)` before proceeding to process the retrieved item.

To support accurate and detailed evaluation, the semaphore version includes an event logging system that records the behavior of each thread. For every attempted and completed synchronization operation, the system records a high-resolution timestamp, the thread index, the thread's role, and an event type, which includes requests (P_REQ, C_REQ), completions (P_DONE, C_DONE), and blocking/unblocking events (BLOCK, UNBLOCK). These logs, stored in memory and then written to a CSV file at the experiment's conclusion, provide the complete, granular data necessary for post-run analysis of throughput, thread fairness (using Jain's index), average waiting times, and critical correctness checks, such as verifying buffer invariants and matching produced and consumed item IDs.

The experiment's execution flow is controlled by the user-specified input, including the number of producer threads, consumer threads, and the fixed total number of items each producer will create. Once all producers have finished generating their real items, the main thread initiates a graceful shutdown using the **Poison Pill** mechanism. It inserts one special POISON_PILL item (ID of -1L) for every

consumer using the standard `buffer_put` routine. Upon retrieving this special item, the consumer thread immediately terminates its operational loop and exits, ensuring the entire system shuts down deterministically without the risk of consumers deadlocking while waiting for non-existent work.

Phase 2 – Monitor Implementation (Katelyn)

The second synchronization method used in this project was the monitor-based producer and consumer system. In this approach, all shared states were contained inside a single monitor structure that controlled how threads interacted with the buffer. The monitor used one mutex to protect the buffer and two condition variables that allowed the system to suspend producers and consumers whenever they could not proceed. This helped maintain a consistent and organized flow of execution between threads.

The monitor relied on a circular buffer that stored item identifiers produced by the system. Access to this buffer always occurred while the mutex was locked. When a producer attempted to add an item, it checked whether the buffer was full. If the buffer had reached capacity, the producer waited on the `notFull` condition variable until a consumer removed an item. When space was available, the producer inserted a new item, updated the circular index, and signaled the `notEmpty` condition variable to allow a consumer to continue. Consumers followed the same controlled sequence. If the buffer was empty, a consumer waited on the `notEmpty` condition variable, removed the next available item when it became available, and signaled `notFull` to notify producers that space had opened.

To support accurate evaluation, the monitor version included event logging that recorded the behavior of each thread. For every attempted and completed operation, the system stored a timestamp, thread index, thread type, event type, and item identifier. These entries were written in a CSV file at the end of each experiment. The logs provided complete information for later analysis of throughput, fairness, waiting time, and correctness.

The monitor version accepted the same command-line parameters as the semaphore version. The user specified the desired number of producer threads, consumer threads, and the total number of items to be processed. Once the system consumed the target number of items, the program completed the experiment and wrote the final log. Using the same input structure for both synchronization techniques allowed us to compare their behavior under identical conditions.

IV. Experimental Results

Phase 1 – Semaphore Implementation

1.1 Experimental Setup:

To study scaling behavior, I used three configurations:

Workload	Buffer capacity N	Producers P	Consumers C	Items/producer	Total real items
Log: 1	100	2	2	50	100
Log: 2	2500	5	5	500	2500
Log: 3	50000	10	10	5000	50000

Log: 4	1000	20	1000	50	1000
Log: 5	100000	20	1000	5000	100000
Log: 6	500000	1000	20	500	500000

1.2 Analysis:

<pre> tigr@kali:~/bin\$./semaphore \$./semaphore Semaphore-based Producer-Consumer (mutex + semaphores) Enter buffer capacity (<= 1000000): 100 Enter number of producers (<= 1000): 2 Enter number of consumers (<= 1000): 2 ===== SUMMARY (Semaphore version) ===== Buffer capacity : 100 Producers / Consumers : 2 / 2 Items per producer : 50 Total real items : 100 Elapsed wall time : 2.002 s Throughput : 49.954 items/s Average producer wait : 0.000 ns Average consumer wait : 0.000 ns Fairness (Jain Index) : Producers : 1.0000 Consumers : 0.5000 Starved producers : 0 Starved consumers : 1 CPU usage (getrusage): User time : 0.002 s System time : 0.002 s Total CPU time : 0.003 s CPU time per item : 0.000033 s/item [Correctness] Buffer invariant violations: 0 [Correctness] Buffer occupancy stayed within [0, 100]. [Correctness] Produced items: 100, Consumed items: 100 [Correctness] No lost or duplicated item IDs detected. Event log written to semaphore_log1.csv (for further analysis). ===== </pre> <p>Log: 1</p>	<pre> tigr@kali:~/bin\$./semaphore \$./semaphore Semaphore-based Producer-Consumer (mutex + semaphores) Enter buffer capacity (<= 1000000): 2500 Enter number of producers (<= 1000): 5 Enter number of consumers (<= 1000): 5 ===== SUMMARY (Semaphore version) ===== Buffer capacity : 2500 Producers / Consumers : 5 / 5 Items per producer : 500 Total real items : 2500 Elapsed wall time : 2.004 s Throughput : 1247.396 items/s Average producer wait : 0.001 ns Average consumer wait : 0.000 ns Fairness (Jain Index) : Producers : 1.0000 Consumers : 0.3995 Starved producers : 0 Starved consumers : 3 CPU usage (getrusage): User time : 0.003 s System time : 0.003 s Total CPU time : 0.006 s CPU time per item : 0.000002 s/item [Correctness] Buffer invariant violations: 0 [Correctness] Buffer occupancy stayed within [0, 2500]. [Correctness] Produced items: 2500, Consumed items: 2500 [Correctness] No lost or duplicated item IDs detected. Event log written to semaphore_log2.csv (for further analysis). ===== </pre> <p>Log: 2</p>	<pre> tigr@kali:~/bin\$./semaphore \$./semaphore Semaphore-based Producer-Consumer (mutex + semaphores) Enter buffer capacity (<= 1000000): 50000 Enter number of producers (<= 1000): 10 Enter number of consumers (<= 1000): 10 ===== SUMMARY (Semaphore version) ===== Buffer capacity : 50000 Producers / Consumers : 10 / 10 Items per producer : 5000 Total real items : 50000 Elapsed wall time : 2.004 s Throughput : 24952.092 items/s Average producer wait : 0.007 ns Average consumer wait : 0.015 ns Fairness (Jain Index) : Producers : 1.0000 Consumers : 0.7379 Starved producers : 0 Starved consumers : 0 CPU usage (getrusage): User time : 0.030 s System time : 0.039 s Total CPU time : 0.069 s CPU time per item : 0.000001 s/item [Correctness] Buffer invariant violations: 0 [Correctness] Buffer occupancy stayed within [0, 50000]. [Correctness] Produced items: 50000, Consumed items: 50000 [Correctness] No lost or duplicated item IDs detected. Event log written to semaphore_log3.csv (for further analysis). ===== </pre> <p>Log: 3</p>
<pre> tigr@kali:~/bin\$./semaphore \$./semaphore Semaphore-based Producer-Consumer (mutex + semaphores) Enter buffer capacity (<= 1000000): 1000 Enter number of producers (<= 1000): 20 Enter number of consumers (<= 1000): 1000 ===== SUMMARY (Semaphore version) ===== Buffer capacity : 1000 Producers / Consumers : 20 / 1000 Items per producer : 50 Total real items : 1000 Elapsed wall time : 2.424 s Throughput : 412.531 items/s Average producer wait : 0.000 ns Average consumer wait : 0.000 ns Fairness (Jain Index) : Producers : 1.0000 Consumers : 0.6010 Starved producers : 0 Starved consumers : 999 CPU usage (getrusage): User time : 0.015 s System time : 0.418 s Total CPU time : 0.433 s CPU time per item : 0.000433 s/item [Correctness] Buffer invariant violations: 0 [Correctness] Buffer occupancy stayed within [0, 1000]. [Correctness] Produced items: 1000, Consumed items: 1000 [Correctness] No lost or duplicated item IDs detected. Event log written to semaphore_log4.csv (for further analysis). ===== </pre> <p>Log: 4</p>	<pre> tigr@kali:~/bin\$./semaphore \$./semaphore Semaphore-based Producer-Consumer (mutex + semaphores) Enter buffer capacity (<= 1000000): 10000 Enter number of producers (<= 1000): 20 Enter number of consumers (<= 1000): 1000 ===== SUMMARY (Semaphore version) ===== Buffer capacity : 10000 Producers / Consumers : 20 / 1000 Items per producer : 500 Total real items : 10000 Elapsed wall time : 2.442 s Throughput : 3750.349 items/s Average producer wait : 0.078 ns Average consumer wait : 3.178 ns Fairness (Jain Index) : Producers : 1.0000 Consumers : 0.6584 Starved producers : 0 Starved consumers : 214 CPU usage (getrusage): User time : 0.129 s System time : 0.528 s Total CPU time : 0.657 s CPU time per item : 0.000007 s/item [Correctness] Buffer invariant violations: 0 [Correctness] Buffer occupancy stayed within [0, 10000]. [Correctness] Produced items: 10000, Consumed items: 10000 [Correctness] No lost or duplicated item IDs detected. Event log written to semaphore_log5.csv (for further analysis). ===== </pre> <p>Log: 5</p>	<pre> tigr@kali:~/bin\$./semaphore \$./semaphore Semaphore-based Producer-Consumer (mutex + semaphores) Enter buffer capacity (<= 1000000): 500000 Enter number of producers (<= 1000): 1000 Enter number of consumers (<= 1000): 20 ===== SUMMARY (Semaphore version) ===== Buffer capacity : 500000 Producers / Consumers : 1000 / 20 Items per producer : 50 Total real items : 500000 Elapsed wall time : 2.296 s Throughput : 217817.382 items/s Average producer wait : 0.002 ns Average consumer wait : 0.005 ns Fairness (Jain Index) : Producers : 1.0000 Consumers : 0.9356 Starved producers : 0 Starved consumers : 0 CPU usage (getrusage): User time : 0.296 s System time : 0.134 s Total CPU time : 0.430 s CPU time per item : 0.000001 s/item [Correctness] Buffer invariant violations: 0 [Correctness] Buffer occupancy stayed within [0, 500000]. [Correctness] Produced items: 500000, Consumed items: 500000 [Correctness] No lost or duplicated item IDs detected. Event log written to semaphore_log6.csv (for further analysis). ===== </pre> <p>Log: 6</p>

1. Correctness: Race Conditions, Deadlocks, Starvation:

Across all six runs, correctness was maintained. The buffer occupancy consistently stayed within the configured capacity, $[0, N]$ (where $N=100, 2500, 50000, 1000, 100000, 500000$). This means no consumer removed an item from an empty buffer, and no producer inserted an item into a full buffer. So, the buffer invariants were never violated. All six runs printed "No lost or duplicated item IDs detected". This confirms that every item was produced and consumed exactly once, preventing race conditions that lead to data loss or corruption.

Deadlocks: No deadlocks were observed in any of the six stress tests, as all runs were terminated successfully and a summary printed. The **Poison Pill mechanism** used for graceful shutdown ensures consumers do not deadlock waiting for non-existent work. In this code, the poison pill is just a special item value (`POISON_PILL = -1L`) that's never used for real work and is treated as a shutdown signal for consumers. After all producers' finish generating their normal positive-ID

items, the main thread inserts exactly one poison pill into the buffer for each consumer, using the same `buffer_put` logic as real items so all semaphore and mutex rules are respected. Each consumer keeps calling `buffer_get`; when it removes an item, it checks if (`item_id == POISON_PILL`) and, instead of processing it, immediately breaks out of its loop and exits. This guarantees that once there are no more real items, every consumer eventually wakes up, sees its poison pill, and terminates cleanly instead of blocking forever on an empty buffer.

Starvation: While the producers achieved perfect fairness, the consumer side frequently suffered from starvation. The problem was most acute in small and medium workloads, where the OS scheduler, interacting with arbitrary semaphore wakeups, heavily favored some consumer threads over others. For instance, in the smallest run (log 1: 100 items), 1 out of 2 consumers starved. In a run with high consumer contention (log 5, 1,000 consumers), 214 consumers were starved. However, in the largest workloads (log 3 and log 6, 50,000 and 500,000 items respectively), starvation was eliminated, and all consumers handled at least some items. So, the system makes progress and consumes all items, but for the smaller and medium workloads, the OS scheduler and semaphore wakeups cause consumer-side starvation. As a result, some consumer threads never get any items at all.

2. Performance: Throughput and Waiting Time:

Throughput:

Log	Total Items	Elapsed Wall Time	Throughput	Observation
Log 1	100	2.002s	49.95items/s	Lowest throughput
Log 2	2500	2.004s	1247.40items/s	
Log 3	50000	2.004s	24,952.09items/s	
Log 4	1000	2.424s	412.531items/s	High thread counts with low items/producer
Log 5	100000	2.642 s	37,850.349items/s	Very high throughput due to large item count
Log 6	500000	2.296s	217,817.382items/s	Highest throughput overall

System **throughput** scaled significantly with the increase in total items processed¹¹. While the elapsed wall time remained relatively constant across the runs (near 2 seconds) due to fixed overheads, the processing rate increased drastically. The throughput ranged from the lowest in the small run (log 1) at 49.95 items/s to the highest in the largest run (log 6) at 217,817.382 items/s. This demonstrates the system's ability to efficiently handle massive workloads.

Waiting times: Average P/C wait times reported:

Log	Average Producer Wait	Average Consumer Wait	Observation

Log 1	0.000 ms	0.000 ms	
Log 2	0.001 ms	0.000 ms	
Log 3	0.007 ms	0.015 ms	Highest wait times of the initial three runs
Log 4	0.000 ms	0.000 ms	
Log 5	0.078 ms	3.178 ms	Highest consumer wait time
Log 6	0.002 ms	0.005 ms	

Average producer and consumer wait times remain **extremely small (in microseconds)** across most runs, indicating that the buffer is rarely full or empty long enough to cause significant blocking. log 5 exhibits the highest waiting times, particularly for consumers (3.178ms), likely due to the high number of consumers (1000) competing for a relatively smaller number of available items.

3. CPU Usage and Blocking Behavior:

Summary of CPU time, system time and user time:

Log	Total Items	Total CPU Time	CPU Time per Item
Log 1	100	0.003 s	33 μ s/item
Log 2	2,500	0.006 s	2 μ s/item
Log 3	50,000	0.069 s	1 μ s/item
Log 4	1,000	0.433 s	433 μ s/item
Log 5	100,000	0.657 s	7 μ s/item
Log 6	500,000	0.430 s	1 μ s/item

As the workload increases (from 100 to 500,000 items), the average CPU time per item naturally decreases. This trend is primarily due to the amortization of fixed overheads such as thread creation, data structure initialization, and final summary printing over a larger number of items. When the total item count is small, like in log 1 (100 items), each item "inherits" a larger share of this overhead, resulting in a relatively high cost per item (33 μ s/item). Conversely, in the largest run (log 6, 500,000 items), these fixed costs are shared across far more operations, driving the average CPU time per item down to its lowest value 1 μ s/item). The efficient POSIX semaphore implementation utilizes sem_wait to handle blocking conditions (producer finds a full buffer; consumer finds an empty buffer). This mechanism directs the Operating System (OS) to put the thread to sleep instead of having it loop and check the buffer state repeatedly (known as a "busy-wait"). Because sleeping threads do not burn CPU cycles, the processor is freed to run other threads or remain idle. Even with high concurrency, such as in log 5 (20 producers / 1000 consumers), the overall CPU usage stays low, and the program remains efficient, demonstrating the advantage of using OS-level blocking primitives for synchronization.

4. Fairness

Jain's index is used to quantify **how evenly the work is shared among threads**. For a set of non-negative values x_1, x_2, \dots, x_n (here, the number of items handled by each thread), Jain's index is defined as:

$$J = \frac{(\sum_{i=1}^n x_i)^2}{n \cdot \sum_{i=1}^n x_i^2}$$

The value of J is always between 0 and 1. A value of **1** means perfect fairness (all threads did exactly the same amount of work), and values closer to **0** mean the work is very unevenly distributed (a few threads did almost everything while others did little or nothing).

Producers: In all six runs, J_producer is 1.0000. Because each producer is forced to generate the same number of items (50, 500, 5000, 1000, 100000, or 500000). The design guarantees perfect fairness across producers.

Consumers: In sharp contrast, consumer fairness varied drastically, reflecting a significant weakness in the semaphore-based approach's ability to distribute work evenly. The consumer Jain indices and corresponding starvation counts are:

- Small (Log 1): J_consumer = 0.5000 with 1 starved consumer (out of 2).
- Medium (Log 2): J_consumer = 0.3995 with 3 starved consumers (out of 5).
- Large (Log 3): J_consumer = 0.7579 with 0 starved consumers.
- Contention (Log 4): J_consumer = 0.0010 with 999 starved consumers (out of 1000).
- High Threads (Log 5): J_consumer = 0.0584 with 214 starved consumers (out of 1000).
- High Workload (Log 6): J_consumer = 0.9356 with 0 starved consumers.

For smaller and medium workloads (log 1, log 2), and particularly in high-contention scenarios like log 4 and log 5, the J_consumer is extremely low, indicating that most consumer threads did little or no work. A few active consumers handled nearly all the items, while others remained idle or starved. The lowest fairness was observed in log 4 (0.0010), where a vast majority of the 1,000 consumers were starved. Even in the run with 100,000 items (log 5), the high thread counts still resulted in a very low fairness index (0.0584) due to 214 starved consumers. Fairness only improved substantially in the large workload configurations (log 3 and log 6) where the total work was high relative to the number of active threads. In log 6, the J_consumer reached its maximum at 0.9356 with zero starvation, but this remains below the perfect 1.0 index.

Overall, these results clearly demonstrate that while the semaphore solution successfully ensures mutual exclusion and global progress, it fundamentally **fails to guarantee per-thread fairness** on the consumer side. This is attributed to the non-deterministic nature of semaphore wakeups and the OS scheduler's decision, which often heavily favors a few consumers over others.

Phase 2 – Monitor Implementation

1. Results

In this section, the performance and behavioral properties of the synchronization mechanism using the monitor are discussed based on three different experimental settings. In each of the experiments, the same implementation of the bounded buffer is used, and the only difference is the number of producer and

consumer threads and the total number of data items to be processed. The experimental results are in terms of total execution time, system throughput, producer and consumer waiting times, fairness among threads, and correctness of the system.

2. Table of Results

Producers	Consumers	Total Items	Runtimes (sec.)	Throughput (items/sec.)
2	2	50	0.163	307
5	5	500	0.538	929
10	10	5000	2.699	1852
20	1000	1000	0.261	3826
20	1000	100000	26.887	3718
1000	20	500000	70.203	7120

Across all experiments, the monitor-based implementation consistently demonstrated correct synchronization behavior and reliable message delivery under a wide range of producer and consumer configurations. In the smaller and mid-sized setups, 2×2 , 5×5 , and 10×10 , the system remained producer-constrained, meaning that consumers frequently encountered empty-buffer waits while producers almost never blocked. Throughput increased predictably as thread counts and total workload increased, rising from roughly 300 items per second in the smallest case to nearly 1850 items per second in the 10×10 configuration. In all three scenarios, fairness across threads remained high; buffer invariants were preserved, and each item was produced and consumed exactly once.

When the number of consumers was scaled dramatically higher than the number of producers, as in the 20×1000 configurations, the system continued to behave predictably. With 1000 consumers competing for items from only 20 producers, the system remained strongly producer-limited, even when the workload increased from 1000 to 100,000 items. Throughput in both cases remained nearly identical, around 3700-3800 items per second, because of the production rate, not consumption capacity, determined overall performance. Fairness among producers was excellent, while fairness among consumers reflected the natural imbalance caused by having far more consumers than available work. Despite the extreme thread count, the system ran without deadlocks, state corruption, or missed messages.

The final configuration inverted the imbalance, using 1000 producers and only 20 consumers to process 500,000 items. This setup shifted the system into a consumer constrained regime. The buffer remained near full capacity; producers frequently waited on `notFull`, and consumers worked continuously throughout the experiment. This configuration achieved the highest throughput observed, about 7100 items per second, highlighting the system's ability to scale when the consumer pool, rather than the producer pool, limits performance. Both producers and consumers maintained strong fairness, and the system continued to run without errors. Overall, the results demonstrate that the monitor implementation scales reliably and remains correct under both balanced and heavily imbalanced workloads.

2.2 Summary of Observed Trends

Across all experiments, the monitor implementation remained correct, stable, and efficient. In the smaller and mid-sized configurations, the system was clearly producer-constrained: consumers

frequently waited for items while producers rarely blocked, and throughput increased steadily as thread counts and workloads grew. The two trials with one thousand consumers showed the same pattern, adding more consumers did not increase performance because production remained the limiting factor.

The final experiment reversed the imbalance and produced a consumer-constrained system, with one thousand producers generating items faster than twenty consumers could process them. This led to the highest throughput observed and showed that performance depends primarily on which side has fewer effective resources. Across all six configurations, fairness stayed high; synchronization remained correct, and no deadlocks or message errors occurred.

IV. Comparison

In our experiments, both the semaphore-based and monitor-based implementations correctly solved the bounded-buffer producer–consumer problem. In all runs, buffer occupancy stayed within the legal range, so no consumer ever removed an item from an empty buffer, and no producer ever inserted into a full buffer. The ID-tracking logic confirmed that every item produced was consumed exactly once, with no losses or duplicates. Both designs also terminated cleanly: the semaphore version used an explicit “poison pill” to signal consumers to exit, while the monitor version ran until a configured total number of items had been produced and consumed. From a pure safety and robust standpoint, avoiding race conditions, deadlocks, and data corruption, the two approaches are essentially equivalent.

The main practical difference appears in fairness and starvation, particularly on the consumer side. In the semaphore implementation, producers always shared work perfectly because each was assigned a fixed number of items. However, consumers often experienced extreme unfairness: for small and medium workloads, a few consumers handled almost all items while others did little or no work at all, and in highly skewed configurations (many consumers but few items) hundreds of consumer threads effectively starved. Fairness improved only when the total workload was very large, allowing OS scheduling and semaphore wakeups to “average out” over time. The monitor implementation, by contrast, distributed work much more evenly in all tested configurations. Producers remained well balanced, and although consumer fairness naturally reflected producer–consumer imbalances (for example, many more consumers than available items), we did not observe the severe starvation that appeared with semaphores. Centralizing control inside the monitor and using condition variables to manage thread wakeups led to more predictable and equitable sharing of work.

Performance results show another trade-off. The semaphore implementation achieved very high throughputs for large workloads, with per-item CPU cost dropping into the microsecond range as fixed overheads (thread creation, initialization, final reporting) were amortized over many items. Waiting times for producers and consumers were generally low, rising only in heavily contended cases with many consumers. The monitor implementation delivered lower absolute throughput and longer wall-clock times for the largest runs but followed a consistent pattern: throughput increased as the total amount of work and thread counts grew, and bottlenecks shifted logically between producers and consumers depending on which side was constrained. Because the two harnesses measure time slightly differently, the raw numbers are not directly comparable, but qualitatively the semaphore solution behaves like a high-

performance, aggressive baseline, while the monitor solution trades some speed for more stable behavior and better fairness.

From a software engineering perspective, the monitor-based solution is easier to reason with and maintain. All shared states live inside a single abstraction with methods that enforce invariants under one mutex and two condition variables, closely matching textbook monitor patterns. The semaphore implementation exposes lower-level control and can deliver higher throughput, but it requires reasoning multiple semaphores and a mutex scattered through producer and consumer code, which makes it more error-prone and harder to extend. Overall, the results suggest that while the semaphore design is attractive when raw throughput is the primary goal, the monitor design is the better general-purpose choice because it provides similar correctness with significantly better fairness, more predictable scheduling behavior, and cleaner, more maintainable code.