HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
School of Information and Communication Technology

Final Report

Producer and Consumer problem

Course: Operating System – IT3070E
Class Code: 161859
Supervisor: Dr. Do Quoc Huy

Group 2

Name	Student ID	Email
Trần Nguyễn Mỹ Anh	20235474	Anh.TNM235474@sis.hust.edu.vn
Nguyễn Thu Huyền	20235507	Huyen.NT235507@sis.hust.edu.vn
Trần Lê Hạ Đan	20235483	Dan.TLH235483@sis.hust.edu.vn
Nguyễn Khánh Ly	20235600	Ly.NK235600@sis.hust.edu.vn

Hanoi, January 2026

Mục lục

Producer and Consumer problem

My Anh Tran Nguyen
20235474
Anh.TNM235474

Huyen Nguyen Thu
20235507
Huyen.NT235507

Ha Dan Tran Le
20235483
Dan.TLH235483

Ly Nguyen Khanh
20235600
Ly.NK235600

1 Introduction

1.1 Overview

The Producer-Consumer problem is a classical synchronization problem in operating systems and concurrent programming that illustrates the challenges of coordinating multiple processes or threads that share a common resource. In this problem, producers generate data items and place them into a shared buffer, while consumers remove and process those items. The key difficulty arises from the need to ensure that producers and consumers access the shared buffer in a safe and orderly manner.

The Producer-Consumer problem is widely used in both theoretical discussions and practical system designs, including task scheduling and data streaming. In the context of Artificial Intelligence and Machine Learning systems, this model appears frequently in data preprocessing pipelines. For example, the CPU acts as a producer by loading images from disk, performing decompression and data augmentation, and placing processed image batches into a queue. The GPU acts as a consumer by retrieving these batches to perform model training. Efficient synchronization is critical: If the pipeline is poorly coordinated, the GPU may remain idle while waiting for data from the CPU, leading to significant waste of computational resources.

1.2 Contribution

This report presents a systematic study of the Producer-Consumer problem through the design and evaluation of three distinct synchronization approaches: busy waiting, condition variables, and semaphores. Each approach represents a different level of abstraction and efficiency in handling concurrent access to shared resources, allowing for a comprehensive comparison of their strengths and limitations. To ensure transparency and facilitate the reproducibility of our experimental results, the complete source code and implementation details are publicly available at: https://github.com/myanhtrannguyen/OperatingSystem_Project.

2 Problem Analysis

2.1 Problem Statement

The Producer-Consumer problem consists of two types of processes or threads: producers and consumers (Distributed Systems Archive, 2024; GP Coder, 2019). Producers are responsible for generating data items, while consumers process or use those items. Communication between producers and consumers occurs through a shared buffer, which temporarily stores produced items until they are consumed (Distributed Systems Archive, 2024). Both producers and consumers may execute concurrently and independently of one another.

The shared buffer is the critical resource in the producer-consumer problem and must be accessed safely by multiple producers and consumers (GP Coder, 2019). Mutual exclusion is required so that only one process or thread can modify the buffer at a time, preventing data corruption (Silberschatz et al., 2018).

In the bounded-buffer variant of the problem, buffer usage is constrained by its capacity. Producers must wait when the buffer is full to avoid overflow, while consumers must wait when the buffer is empty to avoid underflow (Distributed Systems Archive, 2024; GP Coder, 2019).

2.2 Potential Issues

2.2.1 Concurrency Issues

Without appropriate synchronization mechanisms, the Producer-Consumer problem is susceptible to several concurrency-related issues.

Race conditions may occur when multiple threads concurrently access and modify shared variables, such as buffer indices or item counters, resulting in unpredictable and incorrect behavior (Silberschatz et al., 2018). Unsynchronized access can cause data inconsistency, where items may be overwritten, duplicated, or lost before being properly consumed (GP Coder, 2019).

Other problems can also arise, including deadlock, in which producers and consumers wait indefinitely for conditions that never become true (Silberschatz et al., 2018).

Starvation is another potential issue, where certain producers or consumers are repeatedly denied access to the shared buffer due to unfair scheduling or flawed synchronization logic.

2.2.2 Performance Issues

In addition to correctness, performance is a critical concern in the Producer-Consumer problem (Distributed Systems Archive, 2024).

Inefficient synchronization techniques can lead to excessive CPU utilization, particularly in solutions that rely on busy waiting, where threads continuously check buffer conditions instead of blocking (Silberschatz et al., 2018). Such behavior wastes processing resources and reduces overall system efficiency.

Synchronization overhead, including frequent context switches and unnecessary wake-ups, can further degrade performance. As the number of producers and consumers increases, scalability becomes a key issue (Distributed Systems Archive, 2024).

2.3 Problem Objectives

A correct solution to the Producer-Consumer problem must satisfy several objectives. It must ensure mutual exclusion during buffer access, prevent buffer overflow and underflow, and avoid deadlock and starvation (GP Coder, 2019). Additionally, the solution should make efficient use of CPU resources and support multiple producers and consumers operating concurrently, ensuring the scalability of the system (Distributed Systems Archive, 2024).

This analysis establishes the challenges inherent in the Producer-Consumer problem, motivates the use of different synchronization mechanisms to address them effectively, and forms the basis for evaluating the synchronization approaches discussed in the subsequent section.

3 Synchronization Mechanism

When multiple threads access a shared resource such as a buffer, the code segment that reads from or writes to that resource is referred to as a critical section. To prevent race conditions and data corruption, only one thread should be allowed to execute a critical section at any given time. This requirement is known as mutual exclusion.

In addition to mutual exclusion, the Producer-Consumer problem requires condition synchronization. Producers and consumers must coordinate based on the state of the buffer: producers must wait when the buffer is full, and consumers must wait when the buffer is empty. These constraints depend on logical conditions rather than access control.

3.1 Busy Waiting

Busy waiting is a synchronization technique in which a thread repeatedly checks a shared condition while remaining active on the CPU. In this approach, a producer continuously tests whether the buffer has available space, and a consumer continuously tests whether the buffer contains data.

Although busy waiting is simple to implement and helps illustrate fundamental concurrency problems, it is inefficient because it consumes CPU resources even when no useful work can be performed. Furthermore, without proper mutual exclusion, busy waiting does not prevent race conditions.

3.2 Blocking Synchronization and Condition Variables

Blocking synchronization allows threads to suspend execution when a required condition is not met and to resume only when another thread signals that the condition may have changed. Condition variables provide this

functionality by enabling threads to wait and be notified based on shared state.

Condition variables are typically associated with a lock that ensures mutual exclusion while checking and modifying shared data. When a thread calls a wait operation, it atomically releases the lock and enters a blocked state. When another thread signals the condition, waiting threads are awakened and can safely recheck the condition before proceeding. This mechanism improves CPU efficiency and correctness compared to busy waiting.

3.3 Semaphores

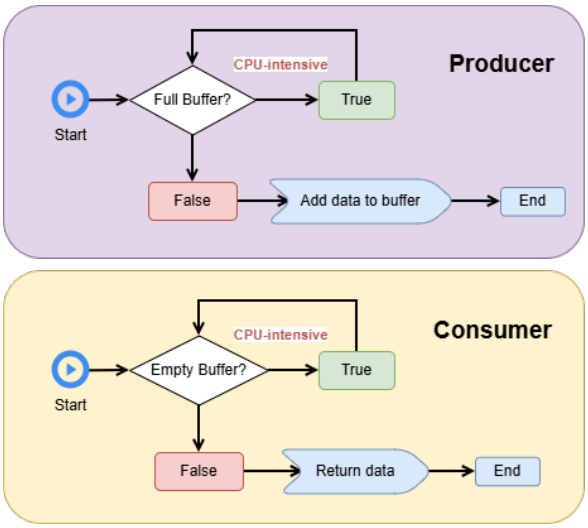
Semaphores are synchronization primitives that use integer counters to control access to shared resources. A semaphore supports two atomic operations: wait (decrement) and signal (increment). When the semaphore value is zero, threads attempting to wait are blocked until another thread signals.

In the Producer-Consumer problem, semaphores are commonly used to track the number of empty buffer slots and the number of full slots. Combined with a mutex for mutual exclusion, semaphores provide an efficient solution that enforces both resource availability and safe access to the shared buffer.

4 Algorithm Design

4.1 Method 1: The Busy Waiting Technique

4.1.1 Idea and working principle



Hình 1: Pipeline of Busy Waiting for Producer and Consumer

This represents the naive approach and the simplest method for solving the Producer-Consumer problem. The core concept relies on natural sequential logic:

- Producer: Prior to production, it continuously queries: "Is the buffer full?" If it is full, the Producer waits and repeatedly re-queries until space becomes available.
- Consumer: Prior to consumption, it continuously queries: "Is the buffer empty?" If the buffer is empty, the Consumer waits and repeatedly re-queries until data is available.

This method does not utilize complex synchronization mechanisms provided by the Operating System (such as Semaphores or Monitors) but relies solely on simple conditional check loops.

4.1.2 Implementation strategy

To implement this method, a fixed-size Buffer is shared between the two processes.

- Employ a counter variable or a function to check the size (count or size) to determine the current state of the Buffer.

- Utilize an empty while loop (a do-nothing loop) to hold the process execution flow while the condition remains unsatisfied.

4.1.3 Algorithmic analysis

Below is the pseudo-code describing the algorithm, isolating the logic from specific programming languages:

Algorithm 1: Busy Waiting

```
Input  : Shared Buffer B with capacity N
Output : Data transfer from Producer to Consumer
// Global variable: count = current number of items
1  Function Producer():
2      while True do
3          item ← produce_item();
           // Busy Wait: Loop continuously while buffer is full
4          while count == N do
               // Do nothing, just burn CPU cycles
5          B[count] ← item;
6          count ← count + 1;
7  Function Consumer():
8      while True do
           // Busy Wait: Loop continuously while buffer is empty
9          while count == 0 do
               // Do nothing
10         item ← B[count − 1];
11         count ← count − 1;
12         process_item(item);
```

Code Analysis: The line While (buffer.size == ...): Do nothing is the crux of "Busy Waiting". The CPU is compelled to execute this comparison instruction millions of times per second merely to receive a True result (indicating the buffer remains full/empty), causing massive computational resource waste without generating any value.

4.1.4 Implementation details

To demonstrate the Busy Waiting technique, we implemented a Python program using the threading module without any advanced synchronization primitives like Semaphores or Conditions.

Input Data:

- Shared Resource: A simple Python list buffer = [] acting as the storage.
- Constraints: $\text{BUFFER}_SIZE = 5$.
- Control flags: A global variable count is used to track the number of items.

Key functions Logic:

- while len(buffer) == LIMIT: pass: This is the core implementation of busy waiting. The Producer thread enters this loop and continuously executes the pass statement (do nothing) as long as the buffer is full.
- CPU Utilization: Since the thread never yields execution to the OS (it never sleeps), this loop consumes 100% of the allocated CPU time slice, leading to high processor usage even when no items are being produced.
- Lack of Atomic Locks: In this naive implementation, we intentionally omitted threading.Lock. This demonstrates that without mutual exclusion, checking the buffer size and appending data are not atomic operations, leading to potential race conditions (e.g., IndexError when popping from an empty list).

Program output:

The console logs show rapid switching between "Checking..."states.

Observation: When running this method, the system fan speed may increase due to high CPU load.

4.1.5 Evaluation

Strengths:

- Simplicity and Ease of Implementation: The logic is highly intuitive and does not require deep knowledge of OS APIs (such as pthreads or java.util.concurrent).
- No Context Switch Overhead: In very rare scenarios (where the wait time is extremely short), busy waiting can technically be faster than putting a thread to Sleep and subsequently waking it up.

Weaknesses (Critical/Fatal Flaws):

- CPU waste: This is the most significant drawback. The process occupies 100% of CPU resources solely to check an unchanging condition. This contradicts the goal of system performance optimization.

- Race condition:

  - The method in the example above has not implemented any Locking/Mutex mechanisms.

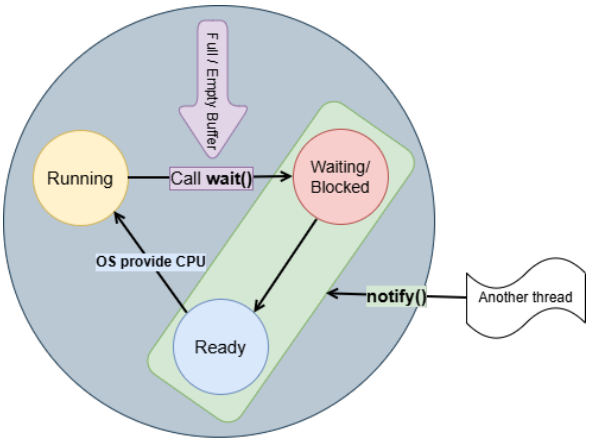
  - According to the analysis, if thread safety is not guaranteed, when multiple Producers simultaneously observe buffer.size < 5 and attempt to add data at the same moment, data corruption or index misalignment will occur.

- Not calable: As the number of Producers/Consumers increases, resource contention and CPU waste increase exponentially, potentially causing system hangs or high latency.

Conclusion for this section: The Busy Waiting method serves to clarify the logical workflow of the Producer-Consumer problem; however, it is infeasible for real-world systems due to performance and data safety issues. Therefore, it is necessary to transition to solutions utilizing Operating System mechanisms such as Monitors (Method 2).

4.2 Method 2: Sleep & Wakeup mechanism (Condition variables)

4.2.1 Idea and working principle



Hình 2: Pipeline of Condition Variables in one Thread for Producer and Consumer

To address the critical flaw of CPU wastage in the Busy Waiting approach, Method 2 introduces a Sleep & Wakeup mechanism (often referred to as the Monitor pattern).

- Core concept: Instead of constantly checking the buffer status, a thread should voluntarily relinquish the CPU and enter a "Waiting"(or Blocked) state if it cannot proceed.

- Producer's logic: If the buffer is full, the Producer goes to sleep (wait). It remains asleep until the Consumer removes an item and wakes it up (notify).

- Consumer's logic: Similarly, if the buffer is empty, the Consumer goes to sleep. It waits for the Producer to add data and wake it up.

- Synchronization: This method utilizes a Condition Variable (often paired with a Lock/Mutex) to atomically handle the locking and waiting process, ensuring thread safety.

4.2.2 Implementation strategy

In this implementation, we utilize a Monitor object (represented by threading.Condition in Python or synchronized block in Java).

- Locking: Before accessing the shared buffer, the thread must acquire a lock.

- Waiting: If the condition is not met (e.g., buffer is full), the thread calls wait(). This action releases the lock and pauses the thread's execution.
- Signaling: When a thread changes the buffer's state (e.g., adds an item), it calls notify() (or signal) to wake up sleeping threads.

4.2.3 Algorithmic analysis

Below is the pseudo-code describing the algorithm, isolating the logic from specific programming languages:

Algorithm 2: Sleep & Wakeup (Condition Variables)

Input : Buffer B, Monitor Lock L, Condition Variable CV

```
1  Function Producer():
2      while True do
3          item ← produce_item();
4          AcquireLock(L);
           // Enter Critical Section
5          while size(B) == N do
               // Release lock L and go to sleep
6              Wait(CV);
7          push(B, item);
           // Wake up sleeping Consumer
8          Notify(CV);
9          ReleaseLock(L);
10 Function Consumer():
11     while True do
12         AcquireLock(L);
           // Enter Critical Section
13         while size(B) == 0 do
               // Release lock L and go to sleep
14             Wait(CV);
15         item ← pop(B);
           // Wake up sleeping Producer
16         Notify(CV);
17         ReleaseLock(L);
18         process_item(item);
```

Code Analysis:

- Wait(condition): This is the key difference from Method 1. Instead of looping, the thread stops execution here, saving CPU resources.
- Signal(condition): This command ensures that if the other party is sleeping, they will be alerted that the state has changed (e.g., from Empty to Not Empty).

4.2.4 Implementation details

For this method, we transitioned from polling to a notification-based model using Python's Monitor pattern.

Input data:

- Synchronization Object: threading.Condition(). This object acts as both a Lock (Mutex) and a waiting room for threads.
- Shared Resource: The same buffer list with size 5.

Key functions Logic:

- with condition:: This statement automatically acquires the underlying lock before entering the critical section and releases it upon exit. It ensures Mutual Exclusion.
- condition.wait():
  - Purpose: When the buffer is full (for Producer) or empty (for Consumer), this function is called.
  - Behavior: It atomically releases the lock and puts the thread into a "Blocked"state. The OS removes the thread from the CPU scheduling queue, reducing CPU usage to near zero while waiting.
- condition.notify$_all$() :
  - Purpose: After adding or removing an item, the active thread calls this function.
  - Behavior: It wakes up all threads currently blocked in wait(), allowing them to re-check the buffer condition.

Program output:

The console log clearly shows threads pausing and resuming:

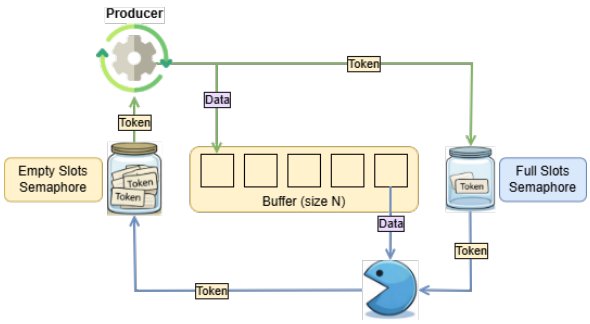
- [Producer] Buffer Full. Waiting... (Thread stops here).
- [Consumer] Consumed item. Notifying...
- [Producer] Woke up. Produced item.

4.2.5 Evaluation

- Strengths:
  - CPU Efficiency: Threads do not consume processor cycles while waiting. They are moved to the "Blocked"queue by the Operating System, allowing other processes to run.
  - Thread Safety: The use of Locks/Mutexes prevents Race Conditions. Only one thread can modify the buffer at a time.
- Weaknesses:
  - Context Switch Overhead: Putting a thread to sleep and waking it up involves a "Context Switch," which has a small cost. However, this is negligible compared to the cost of Busy Waiting.
  - Logic Complexity: Developers must carefully handle the wait and notify placement to avoid "Deadlocks" (where both threads sleep forever) or "Lost Wakeups."
  - Spurious Wakeups: In some OS implementations, a thread might wake up without a signal. Therefore, the condition must always be checked inside a While loop, not an If statement.

4.3 Method 3: Semaphore & Mutex (The Optimized Solution)

4.3.1 Idea and operating principle



Hình 3: Pipeline of Semaphores for Producer and Consumer

This method employs Semaphores, a fundamental synchronization primitive in Operating Systems (originally proposed by Dijkstra). Unlike Method 2 (Monitors), which focuses on "waiting for a condition,"Semaphores focus on "managing permits"to access resources.

- Core Concept: We treat the buffer slots as a pool of resources.
  - Empty Slots: Resources required by the Producer.
  - Full Slots (Data): Resources required by the Consumer.
- Decoupling: As mentioned in the introduction (Article 1), this model allows Producers and Consumers to operate independently. They don't need to know each other's state; they only interact with the Semaphores.
- Mechanism:
  - Counting Semaphores: Used to track the number of available items and empty spaces.
  - Mutex (Mutual Exclusion): Used solely to protect the critical section (the actual push/pop operations) to ensure thread safety.

4.3.2 Implementation strategy

To implement this, we require three distinct synchronization primitives:

1. empty_slots (Counting Semaphore): Initialized to BUFFER_SIZE. It tracks how many spaces are left for the Producer to fill.
2. full_slots (Counting Semaphore): Initialized to 0. It tracks how many items are ready for the Consumer to take.
3. mutex (Lock/Binary Semaphore): Initialized to 1. It ensures that only one thread modifies the buffer structure at a time.

Operations:

- Wait (P operation / Acquire): Decrements the semaphore count. If the count is 0, the thread blocks.
- Signal (V operation / Release): Increments the semaphore count. If threads are blocked, one is woken up.

4.3.3 Algorithm analysis (Pseudo-code)

The following pseudo-code demonstrates the standard implementation used in Operating Systems.

Algorithm 3: Semaphores (Optimized Solution)

Input : Semaphores: $Empty$ (init N), $Full$ (init 0), $Mutex$ (init 1)

```
1  Function Producer():
2      while True do
3          item ← produce_item();
           // Step 1: Wait for an empty slot
4          Wait(Empty);
           // Step 2: Enter Critical Section (Exclusive Access)
5          Wait(Mutex);
6          add_to_buffer(item);
7          Signal(Mutex);
           // Step 3: Signal that a new item is available
8          Signal(Full);
9  Function Consumer():
10     while True do
           // Step 1: Wait for available data
11         Wait(Full);
           // Step 2: Enter Critical Section
12         Wait(Mutex);
13         item ← remove_from_buffer();
14         Signal(Mutex);
           // Step 3: Signal that a slot is now empty
15         Signal(Empty);
16         process_item(item);
```

Code Analysis:

- Separation of Concerns: The Wait(empty_slots) and Signal(full_slots) pair in the Producer manages the flow control, while Wait(mutex) handles data integrity.
- No spurious wakeups: Unlike Method 2 (Condition Variables), Semaphores generally do not suffer from spurious wakeups, making the logic cleaner (no While loop needed for the lock itself, though the semaphore handles the blocking).

4.3.4 Implementaion details

This is the optimized solution where we manage permits to access the buffer, decoupling the Producer and Consumer logic completely.

Input data:

- empty_slots: A threading.Semaphore initialized to BUFFER_SIZE (5). Represents the number of free spaces.
- full_slots: A threading.Semaphore initialized to 0. Represents the number of items ready for consumption.
- mutex: A threading.Lock() to protect the data integrity of the list.

Key functions Logic:

- empty$_s lots.acquire()$:
  - Used by the Producer. It attempts to decrease the empty count.
  - Blocking Logic: If the count is 0 (buffer full), the OS blocks the thread immediately. No while loop is needed to check the condition.
- full_slots.release():
  - Used by the Producer after adding an item. It increments the item count, effectively signaling the Consumer that data is available.
- mutex.acquire() / release():
  - These wrap strictly around the buffer.append() and buffer.pop() operations. This ensures that the "Critical Section" is as short as possible, maximizing concurrency.

Program output:

- The flow is smooth and strictly ordered.
- Scenario: If the Producer is faster, the logs will show [Producer] Waiting for empty slot... and it will strictly wait until a [Consumer] Consumed... log appears.
- Unlike Method 1, there is no CPU waste. Unlike Method 2, there is no risk of "spurious wakeups" or complex condition checking logic.

4.3.5 Evaluation

- Strengths:
  - Scalability: This is the most scalable solution. As noted in Article 1, this pattern allows adding multiple Producers and Consumers without changing the code logic. The Semaphore queue handles the ordering of multiple threads efficiently.
  - Deadlock Prevention: By strictly ordering the Wait operations (Wait for Space -> Wait for Mutex), we avoid standard deadlock scenarios.
  - Industry Standard: This architecture mirrors how Message Queues (like RabbitMQ or Kafka) and AI Data Loaders (like in PyTorch/TensorFlow) operate internally.
- Weaknesses:
  - Implementation Complexity: The logic is abstract. A common mistake is swapping the order of Wait(empty_slots) and Wait(mutex), which causes an immediate Deadlock (the Producer holds the mutex while waiting for space, but the Consumer needs the mutex to make space).
  - Debugging Difficulty: Errors in Semaphore counting can lead to logic bugs that are hard to reproduce.

The semaphore-based solution offers a robust and scalable approach to solving the Producer–Consumer problem.

5 Test Cases

To verify the correctness and performance of the proposed solutions, we conducted tests using the Python threading module on a standard laptop environment (MPS: Macbook Air M1). We focused on three critical scenarios:

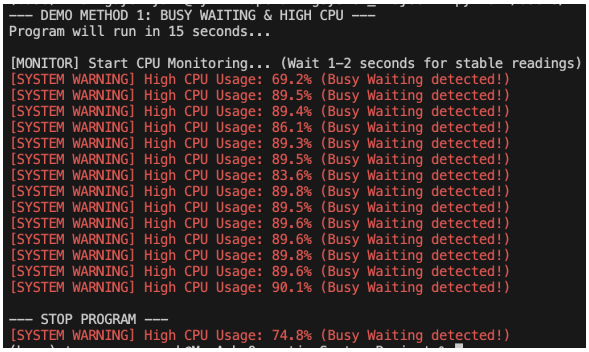
5.1 Scenario 1: Buffer overflow (Producer is faster than Consumer)

Configuration: Buffer size $N = 2$. Producer sleep time $= 0.1s$. Consumer sleep time $= 1.0s$.

Expected behavior: The Producer should fill the buffer quickly and then enter a "Waiting" state. It must not overwrite existing data.

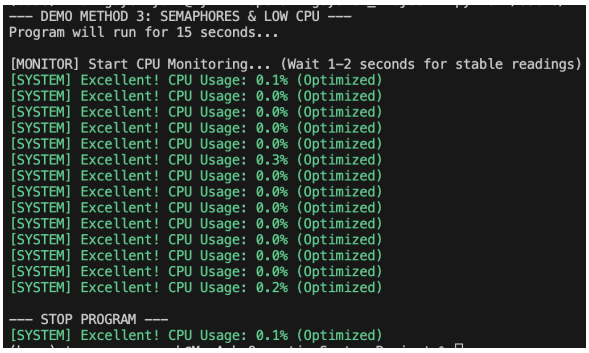
Result:

Busy Waiting (Method 1): CPU usage spiked to 90.1% as the Producer loop checked the condition continuously.



Hình 4: CPU usage - Method 1

Semaphores (Method 3): The Producer paused efficiently. Console log showed: [Producer] Waiting for empty slot....



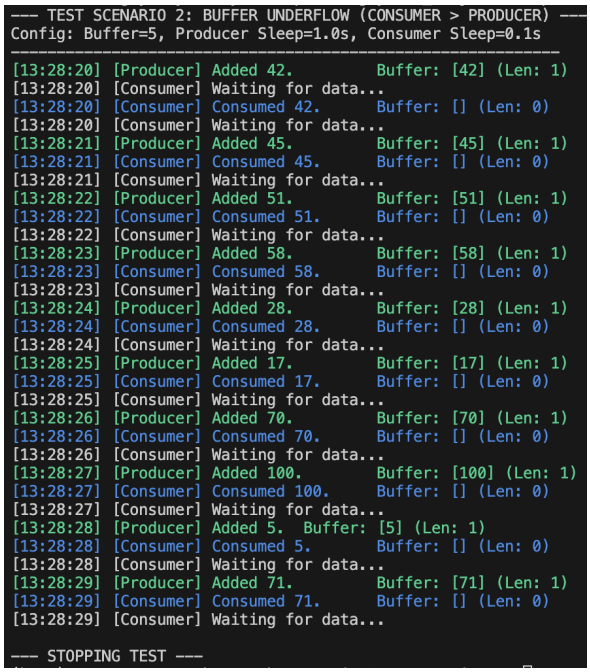
Hình 5: CPU usage - Method 3

5.2 Scenario 2: Buffer underflow (Consumer is faster than Producer)

Configuration: Buffer size $N = 5$. Producer sleep time $= 1.0s$. Consumer sleep time $= 0.1s$.

Expected behavior: The Consumer should empty the buffer and then wait. It must not crash or retrieve None/garbage data.

Result: The Consumer paused correctly. Console log showed: [Consumer] Waiting for data....



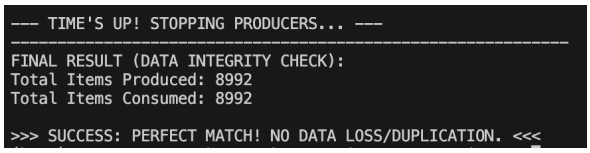
Hình 6: Buffer underflow

5.3 Scenario 3: Concurrency Data integrity

Configuration: 5 Producers and 5 Consumers operating simultaneously.

Expected behavior: All items produced must be consumed exactly once. No items should be lost or duplicated.

Result: Using method 3 (Semaphores + Mutex), the total count of produced items matched the total count of consumed items perfectly after a 60-second run.



Hình 7: Concurrency Data integrity

5.4 Scenario 4: Monitor logic verification (Sleep Wakeup)

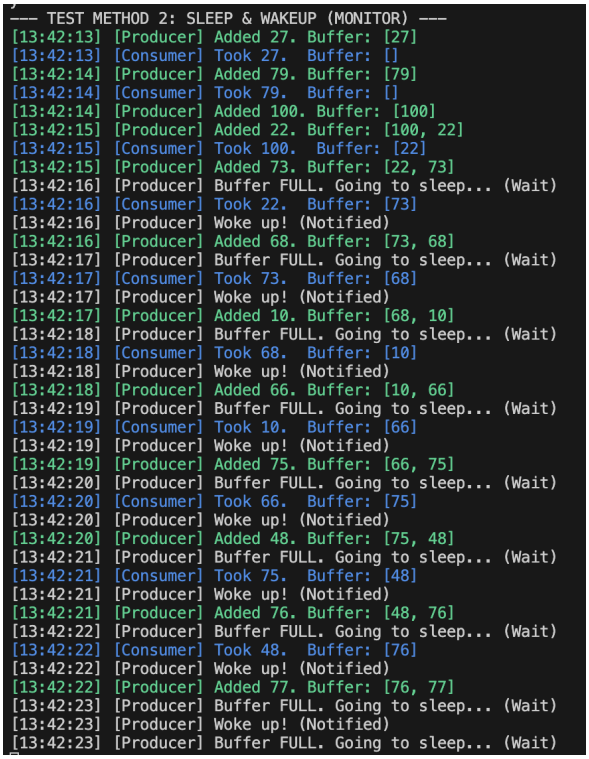
Objective: Verify that threads correctly enter the "Waiting"state when conditions are not met and resume execution only upon notification.

Configuration:

- Buffer Size: $N = 2$.
- Producer speed $>$ Consumer speed (to force Buffer Full).

Expected behavior: The Producer log must explicitly show a transition from "Going to sleep"to "Woke up"only after a Consumer action.

Result: The console log confirmed the behavior. The Producer paused when the buffer reached size 2 and resumed immediately after the Consumer removed an item.



Hình 8: Monitor logic verification

6 Conclusion

This report examined the Producer–Consumer problem through three synchronization techniques: Busy Waiting, Sleep–Wakeup using condition variables, and Semaphore-based synchronization. The Busy Waiting method demonstrates the basic interaction logic but suffers from severe CPU waste and race conditions, making it unsuitable for practical systems. The Sleep–Wakeup approach improves efficiency by allowing threads to block instead of continuously checking conditions, while also ensuring mutual exclusion through locks. However, it requires careful handling of wait and notify operations to avoid synchronization errors.

The Semaphore-based method provides the most effective solution by combining counting semaphores and mutex locks to manage

resource availability and critical sections. This approach eliminates busy waiting, ensures data consistency, and scales well with multiple Producers and Consumers. Although more complex to implement, it offers the best balance between performance, correctness, and scalability. Overall, semaphore-based synchronization is the most suitable technique for real-world Producer–Consumer systems.

References

Distributed Systems Archive. 2024. Producer-consumer model and message queues in distributed systems. Online Article. Accessed: 2026-01-10.

GP Coder. 2019. Vấn đề nhà sản xuất (producer) – người tiêu dùng (consumer) và đồng bộ hóa các luồng trong java. https://gpcoder.com. Accessed: 2026-01-10.

Abraham Silberschatz, Peter B Galvin, and Greg Gagne. 2018. *Operating System Concepts*, 10 edition. Wiley, Hoboken, NJ.

A Appendix

To ensure transparency and provide fair recognition of individual efforts within the project, Table 1 presents a detailed breakdown of task allocation and contributions from the four team members. The contribution categories are divided into three primary pillars: Code (Implement method), Report (comprehensive technical documentation), and Slides (presentation design and content).

Bảng 1: Project contribution details among team members.

No.	Contributor	Contribution Details		
		Code	**Report**	**Slides**
1	**Trần Nguyễn Mỹ Anh**	Method 1	Method 1	Method 1
2	**Nguyễn Thu Huyền**	Method 2	Method 2	Method 2
3	**Trần Lê Hạ Đan**	Method 3	Introduction, Method 3	Introduction, Method 3
4	**Nguyễn Khánh Ly**	Method 3	Problem Analysis, Method 3	Problem Analysis, Method 3