A LOOK TO THE FUTURE

THE PRODUCER – CONSUMER PROBLEM

Course : Operating System

Class ID: 161859

Supervisor: Dr. Đỗ Quốc Huy

OUR TEAM

1. Trần Nguyễn Mỹ Anh – 20235474 - Leader
2. Nguyễn Thu Huyền – 20235507
3. Trần Lê Hạ Đan – 20235483
4. Nguyễn Khánh Ly - 20235600

TABLE OF CONTENTS

1 Introduction

2 Problem Analysis

3 Synchronization Mechanism

4 Algorithm Design

5 Test Cases

6 Conclusion

01

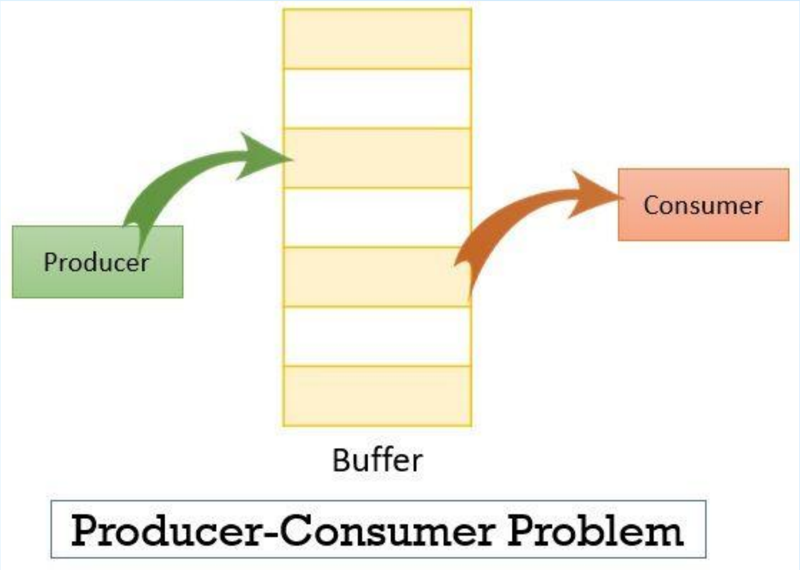
INTRODUCTION

OVERVIEW

Definition: A classical synchronization problem involving multiple processes sharing a common resource.

Components:

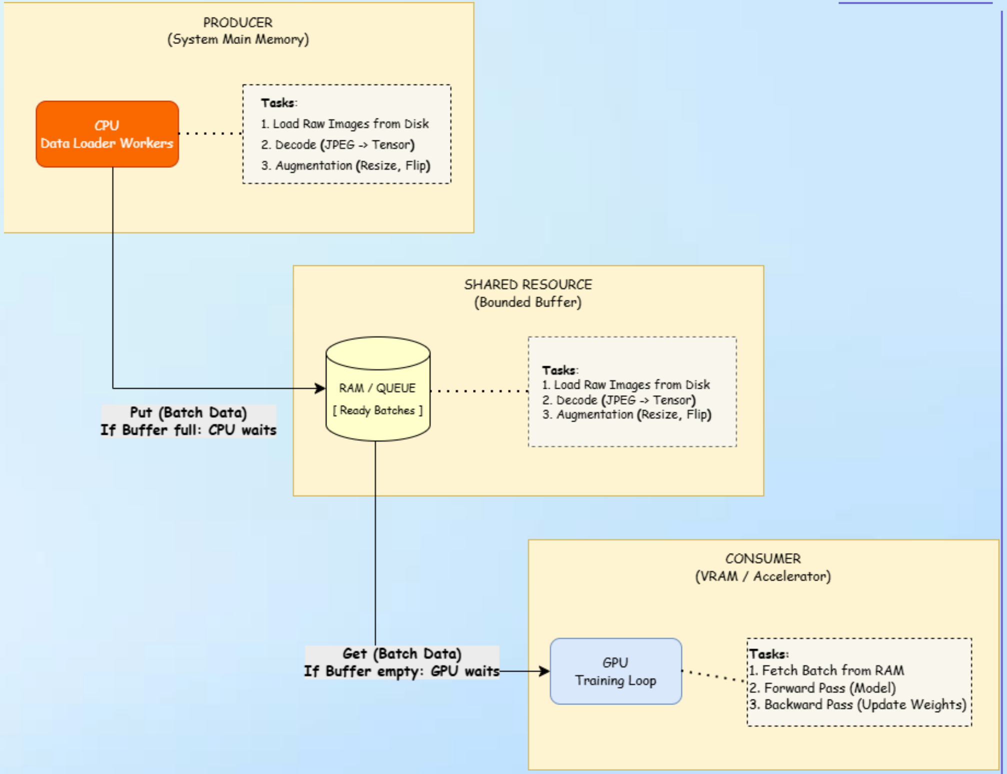
- **Producers:** Generate data items and place them in a shared buffer.
- **Consumers:** Remove and process items from the buffer.
- **Shared Buffer:** The critical resource where data is temporarily stored



OVERVIEW

AI/ML Context:

- **CPU (Producer):** Loads images, performs decompression, and data augmentation.
- **GPU (Consumer):** Retrieves batches for model training.
- **Challenge:** Poor coordination leads to the GPU remaining idle, wasting computational resources



02

PROBLEM ANALYSIS

PROBLEM STATEMENT

- **Key Components:** The system consists of Producers (responsible for generating data items) and Consumers (who process or use those items).
- **Communication Mechanism:** Interaction occurs through a shared buffer that temporarily stores items until they are consumed.
- **Bounded-Buffer Constraints:** In this variant, buffer usage is limited by its capacity.
  - **Overflow:** Producers must wait when the buffer is full.
  - **Underflow:** Consumers must wait when the buffer is empty.
- **Mutual Exclusion:** This is a critical requirement ensuring that only one process or thread modifies the buffer at a time to prevent data corruption

POTENTIAL ISSUES

Concurrency Issues:

- **Race Conditions:** Multiple threads modifying shared variables (counters/indices) simultaneously.
- **Data Inconsistency:** Items may be overwritten or lost.
- **Deadlock & Starvation:** Threads waiting indefinitely for conditions that never become true

Performance issues:

- **CPU Utilization:** Inefficient techniques like Busy Waiting waste processing resources because threads continuously check conditions instead of blocking.
- **Synchronization Overhead:** Frequent context switches and unnecessary wake-ups can significantly degrade system performance.
- **Scalability:** Performance may decrease as the number of producers and consumers increases

POTENTIAL ISSUES

Problem Objectives: A correct and optimized solution must satisfy the following criteria:

1. **Ensure Mutual Exclusion** during all buffer access operations.
2. **Prevent Buffer Overflow and Underflow** through proper coordination.
3. **Avoid Deadlock and Starvation** to ensure system progress.
4. **Optimize CPU Usage** by using blocking mechanisms instead of busy waiting.
5. **Support Scalability** by allowing multiple concurrent producers and consumers.

03

SYNCHRONIZATION MECHANISMS

SYNCHRONIZATION MECHANISMS

Critical Section: The code segment accessing the shared resource.

Key Requirements:

1. **Mutual Exclusion:** Only one thread executes a critical section at a time.
2. **Condition Synchronization:** Coordination based on buffer state (Full/Empty)

THREE APPROACHES

Having established the core difficulties and the need for synchronization in the Producer-Consumer model, we now turn to evaluating three specific mechanisms: Busy Waiting, Blocking Synchronization (Condition Variables), and Semaphores.

1

Busy Waiting

2

Blocking Synchronization and Condition Variables

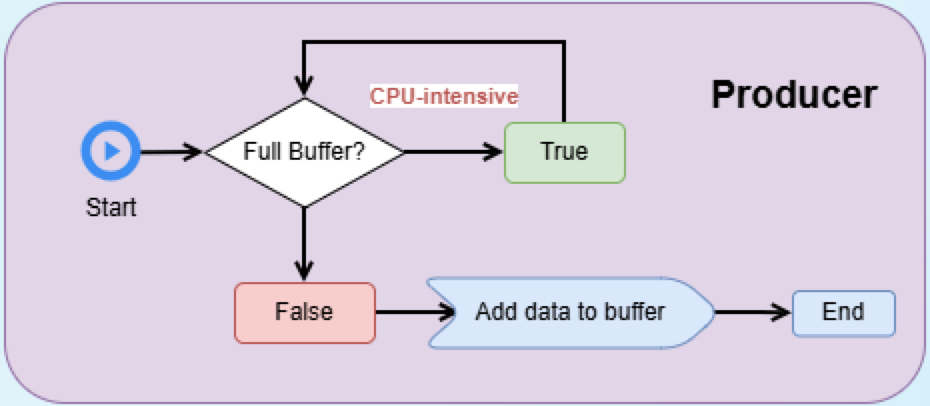
3

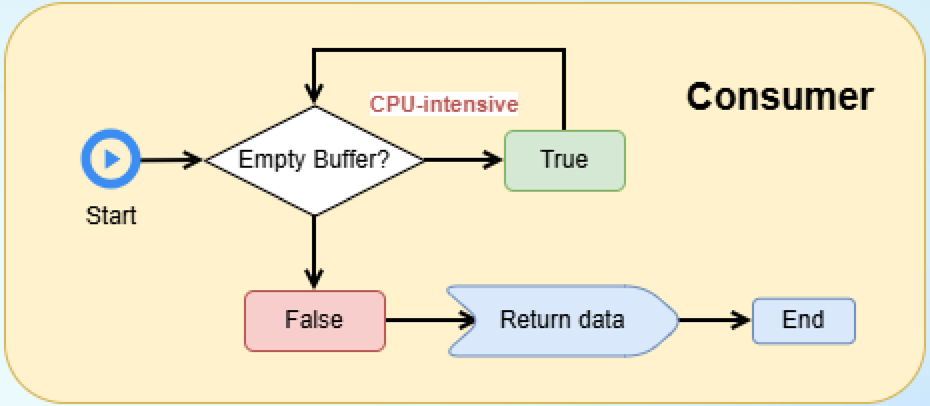
Semaphores

04

ALGORITHM DESIGN

BUSY WAITING





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.

ALGORITHM

Input Data:

- Shared Resource: A simple Python list buffer = [ ] acting as the storage.
- Constraints: BUFFER_SIZE = 5
- Control flags: A global variable **count** is used to check the size to determine the current state of the Buffer.

To implement this method:

- Utilize an **empty while** loop to hold the process execution flow while the condition remains unsatisfied

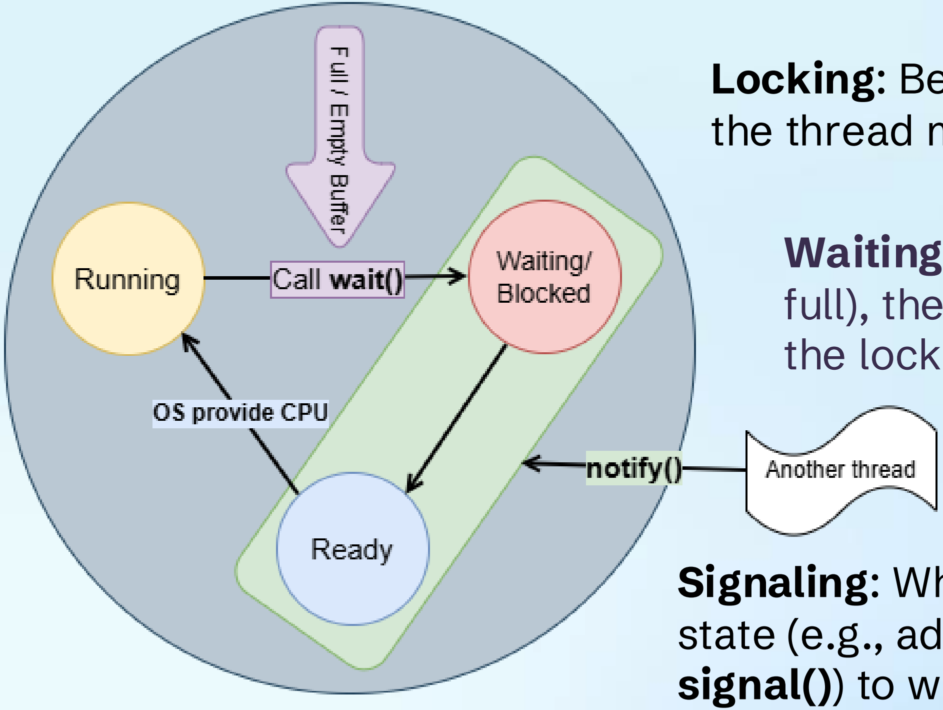
Algorithm 1: Busy Waiting

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

EVALUATION

- **Strengths**
  - Simple and easy to implement
  - No context switch overhead (can be fast in rare short-wait cases)
- **Weaknesses**
  - Severe CPU waste due to constant polling
  - Race conditions may cause data corruption
  - Poor scalability as producers/consumers increase

SLEEP & WAKEUP MECHANISM (CONDITION VARIABLES)



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.

ALGORITHM

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.

Program output:

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

Algorithm 2: Sleep & Wakeup (Condition Variables)

```
Input : Buffer B, Monitor Lock L,
        Condition Variable CV
Function Producer():
    while True do
        item ← produce_item();
        AcquireLock(L);
        // Enter Critical Section
        while size(B) == N do
            // Release lock L and go to
               sleep
            Wait(CV);
        push(B, item);
        // Wake up sleeping Consumer
        Notify(CV);
        ReleaseLock(L);

Function Consumer():
    while True do
        AcquireLock(L);
        // Enter Critical Section
        while size(B) == 0 do
            // Release lock L and go to
               sleep
            Wait(CV);
        item ← pop(B);
        // Wake up sleeping Producer
        Notify(CV);
        ReleaseLock(L);
        process_item(item);
```

ALGORITHM

- **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).

Algorithm 2: Sleep & Wakeup (Condition Variables)

```
Input :Buffer B, Monitor Lock L,
       Condition Variable CV
Function Producer():
    while True do
        item ← produce_item();
        AcquireLock(L);
        // Enter Critical Section
        while size(B) == N do
            // Release lock L and go to
               sleep
            Wait(CV);
        push(B, item);
        // Wake up sleeping Consumer
        Notify(CV);
        ReleaseLock(L);
Function Consumer():
    while True do
        AcquireLock(L);
        // Enter Critical Section
        while size(B) == 0 do
            // Release lock L and go to
               sleep
            Wait(CV);
        item ← pop(B);
        // Wake up sleeping Producer
        Notify(CV);
        ReleaseLock(L);
        process_item(item);
```

EVALUATION

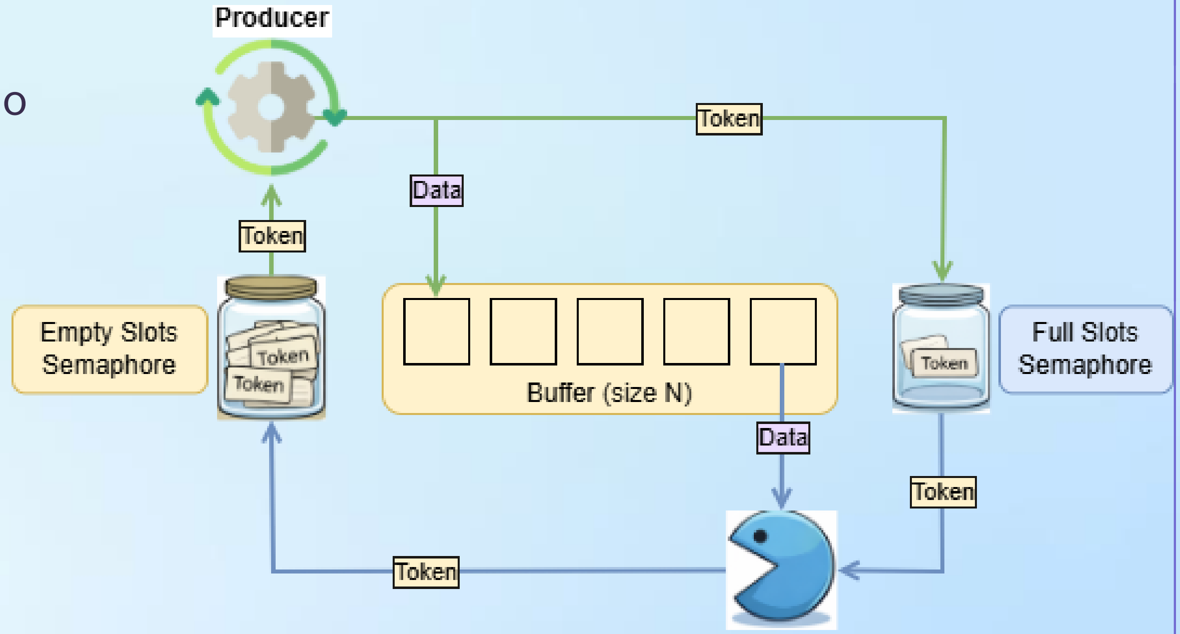
- **Strengths**
  - **CPU efficient:** no busy waiting while threads are blocked
  - **Thread-safe:** mutex/locks prevent race conditions
- **Weaknesses**
  - Context switch overhead when blocking/waking threads
  - More complex logic; incorrect wait/notify may cause deadlocks
  - Wake up without a signal → Require condition checks inside **while,** not **if**

SEMAPHORE & MUTEX

Idea: An optimized solution focusing on managing permits to access resources

Primitives Used:

- ***empty_slots***: Counting semaphore initialized to Buffer Size
- ***full_slots***: Counting semaphore initialized to 0
- ***mutex***: Binary semaphore/lock initialized to 1 for mutual exclusion



Operations: P (Wait) decrements the count/blocks; V (Signal) increments/wakes

ALGORITHM

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.

Program output:

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

Algorithm 3: Semaphores (Optimized Solution)

```
Input   : Semaphores: Empty (init N),
          Full (init 0), Mutex (init 1)
Function Producer():
    while True do
        item ← produce_item();
        // Step 1: Wait for an empty
            slot
        Wait(Empty);
        // Step 2: Enter Critical
            Section (Exclusive Access)
        Wait(Mutex);
        add_to_buffer(item);
        Signal(Mutex);
        // Step 3: Signal that a new
            item is available
        Signal(Full);
Function Consumer():
    while True do
        // Step 1: Wait for available
            data
        Wait(Full);
        // Step 2: Enter Critical
            Section
        Wait(Mutex);
        item ← remove_from_buffer();
        Signal(Mutex);
        // Step 3: Signal that a slot is
            now empty
        Signal(Empty);
        process_item(item);
```

ALGORITHM

- **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, 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).

Algorithm 3: Semaphores (Optimized Solution)

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

```
Function Producer():
    while True do
        item ← produce_item();
        // Step 1: Wait for an empty slot
        Wait(Empty);
        // Step 2: Enter Critical Section (Exclusive Access)
        Wait(Mutex);
        add_to_buffer(item);
        Signal(Mutex);
        // Step 3: Signal that a new item is available
        Signal(Full);

Function Consumer():
    while True do
        // Step 1: Wait for available data
        Wait(Full);
        // Step 2: Enter Critical Section
        Wait(Mutex);
        item ← remove_from_buffer();
        Signal(Mutex);
        // Step 3: Signal that a slot is now empty
        Signal(Empty);
        process_item(item);
```

EVALUATION

- **Strengths**
  - Highly scalable: supports multiple Producers and Consumers efficiently
  - Deadlock-safe with correct semaphore ordering
  - Industry-standard model used in message queues and AI data pipelines
- **Weaknesses**
  - Higher implementation complexity
  - Semaphore misordering may cause deadlock
  - Debugging synchronization errors is difficult

06

TEST CASES

SCENARIO 1: BUFFER OVERFLOW

- **Configuration:**
  - **Buffer size** = 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.

```
--- DEMO METHOD 1: BUSY WAITING & HIGH CPU ---
Program will run in 15 seconds...

[MONITOR] Start CPU Monitoring... (Wait 1-2 seconds for stable readings)
[SYSTEM WARNING] High CPU Usage: 69.2% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.5% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.4% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 86.1% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.3% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.5% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 83.6% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.8% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.5% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.6% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.6% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.8% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 89.6% (Busy Waiting detected!)
[SYSTEM WARNING] High CPU Usage: 90.1% (Busy Waiting detected!)

--- STOP PROGRAM ---
[SYSTEM WARNING] High CPU Usage: 74.8% (Busy Waiting detected!)
```

CPU usage - Method 1

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

```
--- DEMO METHOD 3: SEMAPHORES & LOW CPU ---
Program will run for 15 seconds...

[MONITOR] Start CPU Monitoring... (Wait 1-2 seconds for stable readings)
[SYSTEM] Excellent! CPU Usage: 0.1% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.3% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.0% (Optimized)
[SYSTEM] Excellent! CPU Usage: 0.2% (Optimized)

--- STOP PROGRAM ---
[SYSTEM] Excellent! CPU Usage: 0.1% (Optimized)
```

CPU usage - Method 3

Semaphores :

The Producer paused efficiently.

Console log showed: *[Producer] Waiting for empty slot....*

SCENARIO 2: BUFFER UNDERFLOW

- **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.

SCENARIO 2: BUFFER UNDERFLOW

```
--- TEST SCENARIO 2: BUFFER UNDERFLOW (CONSUMER > PRODUCER) ---
Config: Buffer=5, Producer Sleep=1.0s, Consumer Sleep=0.1s
------------------------------------------------------------
[13:28:20] [Producer] Added 42.         Buffer: [42] (Len: 1)
[13:28:20] [Consumer] Waiting for data...
[13:28:20] [Consumer] Consumed 42.      Buffer: [] (Len: 0)
[13:28:20] [Consumer] Waiting for data...
[13:28:21] [Producer] Added 45.         Buffer: [45] (Len: 1)
[13:28:21] [Consumer] Consumed 45.      Buffer: [] (Len: 0)
[13:28:21] [Consumer] Waiting for data...
[13:28:22] [Producer] Added 51.         Buffer: [51] (Len: 1)
[13:28:22] [Consumer] Consumed 51.      Buffer: [] (Len: 0)
[13:28:22] [Consumer] Waiting for data...
[13:28:23] [Producer] Added 58.         Buffer: [58] (Len: 1)
[13:28:23] [Consumer] Consumed 58.      Buffer: [] (Len: 0)
[13:28:23] [Consumer] Waiting for data...
[13:28:24] [Producer] Added 28.         Buffer: [28] (Len: 1)
[13:28:24] [Consumer] Consumed 28.      Buffer: [] (Len: 0)
[13:28:24] [Consumer] Waiting for data...
[13:28:25] [Producer] Added 17.         Buffer: [17] (Len: 1)
[13:28:25] [Consumer] Consumed 17.      Buffer: [] (Len: 0)
[13:28:25] [Consumer] Waiting for data...
[13:28:26] [Producer] Added 70.         Buffer: [70] (Len: 1)
[13:28:26] [Consumer] Consumed 70.      Buffer: [] (Len: 0)
[13:28:26] [Consumer] Waiting for data...
[13:28:27] [Producer] Added 100.        Buffer: [100] (Len: 1)
[13:28:27] [Consumer] Consumed 100.     Buffer: [] (Len: 0)
[13:28:27] [Consumer] Waiting for data...
[13:28:28] [Producer] Added 5.  Buffer: [5] (Len: 1)
[13:28:28] [Consumer] Consumed 5.       Buffer: [] (Len: 0)
[13:28:28] [Consumer] Waiting for data...
[13:28:29] [Producer] Added 71.         Buffer: [71] (Len: 1)
[13:28:29] [Consumer] Consumed 71.      Buffer: [] (Len: 0)
[13:28:29] [Consumer] Waiting for data...

--- STOPPING TEST ---
```

Result : The Consumer paused correctly.

Console log showed: *[Consumer] Waiting for data....*

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.

```
--- TIME'S UP! STOPPING PRODUCERS... ---
------------------------------------------------------------
FINAL RESULT (DATA INTEGRITY CHECK):
Total Items Produced: 8992
Total Items Consumed: 8992

>>> SUCCESS: PERFECT MATCH! NO DATA LOSS/DUPLICATION. <<<
```

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

- **The most robust and suitable technique for real-world systems, offering the best balance of performance and scalability**

THANK YOU

DO YOU HAVE ANY QUESTIONS?