MY NAME : NIKHIL RAJPOOT
CO23BTECH11015

REPORT

ALGORITHM OF CODE :

Producer-Consumer Problem Algorithms

1. Semaphore-Based Implementation Algorithm

Overall Flow:
1. **Initial Setup:**

Read parameters: buffer size, # producers/consumers, items per thread, delay times

Create buffer of given capacity

Initialize three semaphores:

EmptySemaphore (tracks empty slots, starts at buffer size)

FullSemaphore (tracks filled slots, starts at 0)

LockingSemaphore (mutex for buffer access, starts at 1)

2. Producer Thread Behavior:

For each item to produce:
1. Record start time
2. Wait for empty slot (acquire EmptySemaphore)

3. Get exclusive buffer access (acquire LockingSemaphore)

4. CRITICAL SECTION:
   - Insert item into buffer
   - Update next insertion point
   - Log production details

5. Release buffer access (release LockingSemaphore)
6. Signal new filled slot (release FullSemaphore)
7. Simulate production delay (exponential random time)
8. Record end time and calculate operation duration

After all items:
Calculate average delay per production

3. Consumer Thread Behavior:

For each item to consume:
1. Record start time
2. Wait for filled slot (acquire FullSemaphore)
3. Get exclusive buffer access (acquire LockingSemaphore)

4. CRITICAL SECTION:
   - Remove item from buffer
   - Update next removal point
   - Log consumption details

5. Release buffer access (release LockingSemaphore)
6. Signal new empty slot (release EmptySemaphore)
7. Simulate consumption delay (exponential random time)
8. Record end time and calculate operation duration

After all items:
Calculate average delay per consumption

4. Main Program:

- Validate parameters won't cause deadlock
- Create producer and consumer threads
- Wait for all threads to complete
- Calculate and display average delays
- Clean up resources

2 . locks based - Implementation Algorithm

Overall Flow:
1. **Initial Setup:**
Read same parameters as semaphore version

Create buffer of given capacity

Initialize synchronization primitives:

buffer_mutex (protects buffer access)

file_mutex (protects output file)

not_full condition variable (signals when space available)

not_empty condition variable (signals when items available)

Initialize buffer state counters:

Count (current items in buffer)

next_in (next insertion index)

next_out (next removal index)

2. Producer Thread Behavior:

For each item to produce:

1. Record start time
2. Lock buffer mutex
3. While buffer is full:
   - Wait on not_full condition

4. CRITICAL SECTION:
   - Insert item into buffer
   - Increment item count
   - Update next insertion point
   - Log production details (using file mutex)

5. Signal not_empty (wake waiting consumers)
6. Unlock buffer mutex
7. Simulate production delay (exponential random time)
8. Record end time and calculate operation duration

After all items:

Calculate average delay per production

3. Consumer Thread Behavior:

For each item to consume:

1. Record start time
2. Lock buffer mutex
3. While buffer is empty:
   - Wait on not_empty condition

4. CRITICAL SECTION:
   - Remove item from buffer

- Decrement item count
- Update next removal point
- Log consumption details (using file mutex)

5. Signal not_full (wake waiting producers)
6. Unlock buffer mutex
7. Simulate consumption delay (exponential random time)
8. Record end time and calculate operation duration

After all items:
Calculate average delay per consumption

4. Main Program:

- Validate parameters to prevent deadlock
- Create and start all threads
- Wait for thread completion
- Compute and report statistics
- Clean up synchronization objects

<u>Key Differences Between Approaches</u>:

1. Synchronization Mechanism:

- Semaphore version uses counter semaphores + mutex semaphore
- Mutex version uses condition variables with explicit waiting

2.State Tracking:

- Semaphore version implicitly tracks state through semaphore counts
- Mutex version explicitly maintains Count variable

3.Wakeup Policy:

- Semaphores automatically manage waiting threads
- Condition variables require explicit signaling

4.Implementation Style:

- Semaphore version has more symmetric producer/consumer code
- Mutex version makes buffer state more visible

EDGE CASES WHERE WE HAVE TO TAKE INPUT CAREFULLY

1. **IF nc x cntc > np x cntp**

   Means total consumption of items exceeds total production, Causing at least some consumer to wait indefinitely for an item That will never produce a deadlock.

2. **IF np x cntp - nc x cntc > capacity**

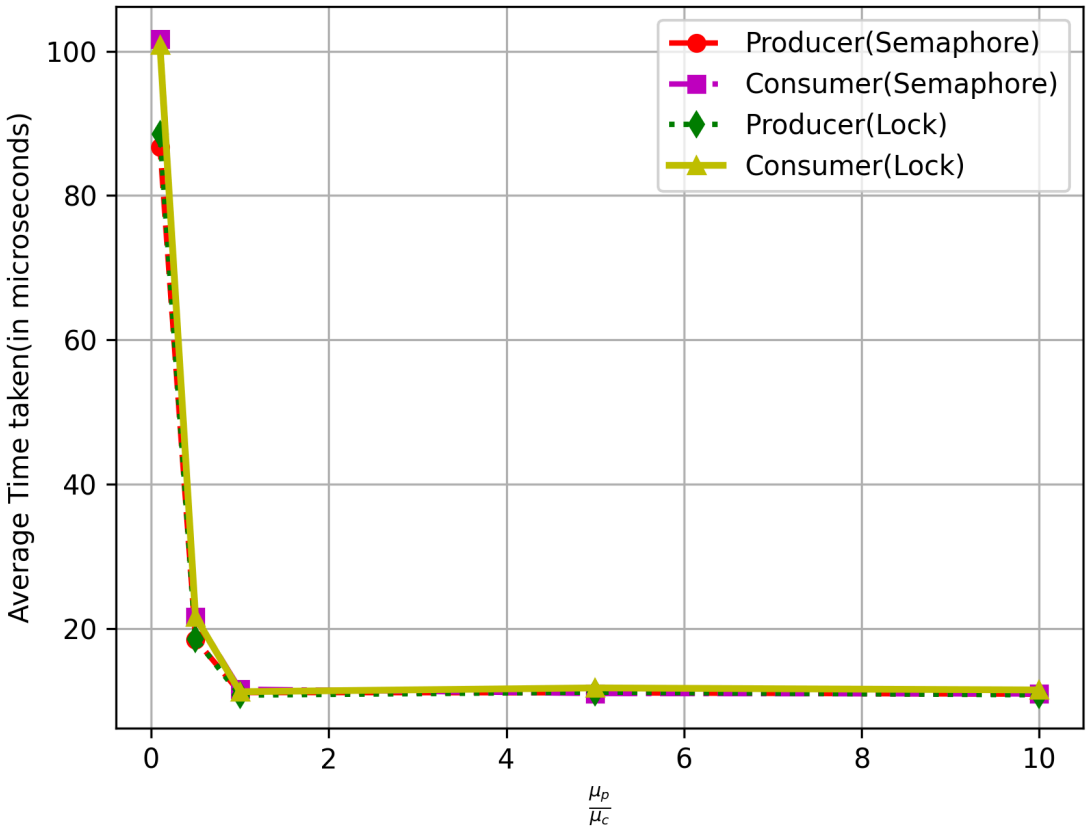
   Means excess produced items cannot be stored in buffer causing Buffer overflow leading producers to block forever waiting for free slot.

EXPERIMENT 1

Keeping μp fix and changing μc and take ratio in X axis

I am running graph for different values on input below one at a line (capacity , np , nc , cntp , cntc, up , uc) values

100 10 10 100 100 10 1
100 10 10 100 100 10 2
100 10 10 100 100 10 10
100 10 10 100 100 10 20
100 10 10 100 100 10 100



OBSERVATION OF EXPERIMENT 1

Effect of the Ratio μp/μc on Average Delay Time

- **μp/μc<1:**
  Consumers are slower:, meaning each consumer takes a longer time to consume an item.

  Buffer Fills Quickly: Because consumption is slow, the producer can fill the buffer more rapidly than it is emptied, causing producers to block (when the buffer is full)

  High Producer Blocking: Once the buffer is full, the producer must wait. This producer blocking inflates overall delay times.Due to which It is taking too much time.

- **μp/μc=1**:

  Equal Production and Consumption: Items are produced and consumed at similar rates.

  Moderate Buffer Usage: The buffer remains neither completely empty nor full for long periods.

  Reduced Delays: Both producer and consumer see relatively balanced wait times, minimizing average delay.

- **μp/μc>1**:

  Producers are slower:, so it takes more time to produce each item.

Consumers Wait for Items: Since items arrive slowly, consumers can end up waiting for new items.

Less Producer Blocking: The buffer is less likely to fill up completely because production is the bottleneck, so producers rarely block.

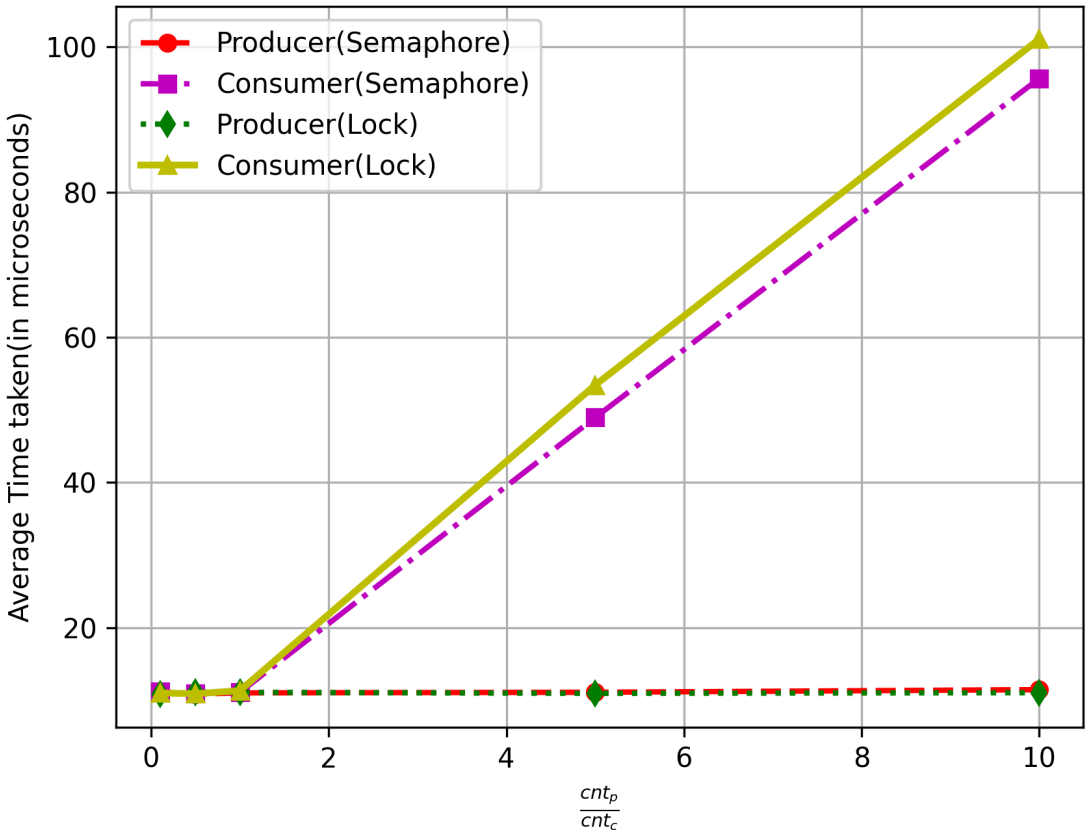
Net Effect on Delay: Consumers wait for items, but the total delay does not grow as drastically as in the case where consumers are very slow (and producers block heavily).

EXPERIMENT 2

Keeping cntp fix and changing cntc and take ratio in X axis

I am running graph for different values on input below
(capacity , np , nc , cntp , cntc, up , uc) values

5000 50 500 100 10 10 10
5000 50 250 100 20 10 10
5000 50 50 100 100 10 10
5000 50 25 100 200 10 10
5000 50 5 100 1000 10 10



OBSERVATION OF EXPERIMENT 2

- **Impact of the Ratio(cntp/cntc) on Average Delay**

- **If Ratio > 1** ((cntp/cntc)>1)

  More Producers Than Consumers: You have many producers (cntp) relative to consumers (cntc)

High Consumer Overhead: Because there are fewer consumer threads but they each need to handle more items, they repeatedly enter/exit critical sections. The overhead from frequent context switches (locking/unlocking or acquire/release) can increase total consumer time.

Producers May Stall Less: With more producers, the buffer can fill quickly, but if consumers are not numerous or fast enough, the buffer may frequently become full, causing producers to block. The net effect depends on how quickly consumers can process items.

- **If Ratio ≈ 1** ($(cntp/cntc) \approx 1$)
    - **Balanced Production and Consumption**: The number of producers is similar to the number of consumers.
    - **Reduced Delays**: The system tends to stay balanced, with fewer long waits on either side. Producers and consumers alternate efficiently, leading to **lower overall average delay**.
- **Ratio < 1** ($(cntp/cntc) < 1$)
    - **Fewer Producers, More Consumers**: There are fewer producers relative to consumers.
    - **Consumers Can Starve**: Because there aren't many producers, the buffer may stay empty more often, causing consumers to wait.
    - **Producer Blocking vs. Consumer Overhead**: The fewer consumer threads may reduce competition (lower overhead

among consumers), but the limited number of producers can fill the buffer slowly—leading to frequent empty-buffer waits for consumers.

COMMON OBSERVATION IN BOTH EXPERIMENTS 1 and 2

1. **Average Consumer Time Is Greater Than Average Producer Time**

REASON –

- Waiting on Production: Consumers depend on producers to fill the buffer. Even if the consumer's own "sleep time" (μc) is low, it can still end up waiting whenever it finds an empty buffer. This waiting time adds to the consumer's total measured time.

- Inclusion of Waiting in Measurements: Your measurements account for the entire time spent in the consumer thread, including both active consumption and any waiting time. Because the consumer must occasionally wait for items, its average time is inevitably higher.

- Producer-Consumer Dependency: Producers only wait if the buffer is full; consumers wait if the buffer is empty. In many scenarios, the buffer empties faster than it fills (especially if μp is not too large), causing the consumer to accumulate waiting time more frequently.

Hence, in most configurations, the consumer's average time is observed to be higher than the producer's average time.

2. Overhead of Semaphore is less than Locks

REASON –

- **Atomic Operations**: acquire and release are often implemented as atomic operations within the OS kernel. This direct, minimal approach reduces the overhead of managing state transitions.

- **Lock + Condition Variable Overhead**: When using locks and condition variables, a thread must:

    1. Acquire the lock

    2. Check a condition (often in a loop)

    3. Potentially wait on the condition variable

    4. Release the lock

- Each of these steps can introduce additional overhead compared to a single atomic semaphore operation.

- **Fewer Context Switches**: Semaphores can reduce the number of context switches and lock/unlock cycles needed. Fewer context switches typically translates to faster performance.

Thus, **semaphore-based solutions** generally have lower average delay and overhead compared to **lock + condition variable** solutions.

END