Tên: Phan Ngọc Hạnh Nhi

MSSV: 2131209002

**Practice Lab 4**

For each of the following problems,

1. Identify the correctness constraints of the problem

2. Specify the conditions that each method must wait for

3. Write down the shared state that you will use to check these conditions

## **Assignment 1 (20 points)**

**1. Identify the correctness constraints of the problem**

* The final value of the shared variable must be 0 after both threads complete execution.
* The program must prevent race conditions to ensure consistency in updates to the shared variable.
* The operations on the shared variable must be atomic, ensuring that no increment or decrement operation is lost due to concurrent execution.
* The solution must avoid deadlocks and allow both threads to complete efficiently.

**2. Specify the conditions that each method must wait for**

* The increase() method must wait if another thread is already modifying the shared variable to prevent concurrent modifications.
* The decrease() method must wait if another thread is already modifying the shared variable to ensure data consistency.
* In synchronized methods, a thread must wait if another thread is executing a synchronized block.
* In ReentrantLock, a thread must wait until it acquires the lock before modifying the shared variable.
* In Semaphore, a thread must wait until the semaphore is available before proceeding with its operation on the shared variable.

**3. Write down the shared state that you will use to check these conditions**

* Shared integer variable (data):
  + Stores the count, initialized to 0.
  + Modified by both threads (increment and decrement).
* Synchronization mechanisms to enforce constraints:
  + synchronized keyword (intrinsic object locking).
  + ReentrantLock (explicit locking for controlled access).
  + Semaphore(1) (ensuring only one thread accesses data at a time).

## **Assignment 2 (20 points)**

**1. Identify the Correctness Constraints of the Problem**

To ensure the correctness of the Thread-Safe Queue (TSQueue), the following constraints must be met:

1. **Preserve FIFO (First-In-First-Out) Order**
   * The removeFirst() method must always remove the earliest inserted element (i.e., the front of the queue).
   * The addLast() method must always insert a new element at the end of the queue.
2. **Ensure Thread Safety**
   * Multiple threads must be able to add and remove elements concurrently without corrupting the queue.
   * The queue must handle simultaneous insertions and removals correctly.
3. **Avoid Race Conditions**
   * Two threads attempting to modify the queue (either addLast() or removeFirst()) at the same time must be synchronized to prevent inconsistencies.
4. **Prevent Deadlocks and Starvation**
   * Threads waiting to access the queue must not be blocked indefinitely.
   * If multiple threads are adding/removing, each thread should eventually proceed in a fair manner.

**2. Specify the Conditions That Each Method Must Wait For**

Each method in the **TSQueue** must wait under certain conditions:

| **Method** | **Waiting Condition** | **Synchronization Mechanism** |
| --- | --- | --- |
| addLast(int x) | Must wait if another thread is modifying the queue. | synchronized, ReentrantLock, or BlockingQueue |
| removeFirst() | Must wait if the queue is empty (cannot remove from an empty queue). | synchronized, ReentrantLock, Semaphore, or Condition (for waiting) |

**Synchronization Approach per Mechanism**

1. **Using synchronized Methods**
   * Only one thread can modify the queue at a time.
   * A thread calling addLast() or removeFirst() must wait if another thread is already executing a synchronized method.
2. **Using ReentrantLock and Condition Variables**
   * A thread must acquire a lock (lock.lock()) before modifying the queue.
   * If the queue is empty, removeFirst() must wait using a condition variable.
3. **Using BlockingQueue (e.g., LinkedBlockingQueue)**
   * A built-in concurrent queue that handles synchronization internally.
   * take() will wait automatically if the queue is empty, and put() will wait if the queue reaches capacity.

**3. Write Down the Shared State That You Will Use to Check These Conditions**

The shared state consists of the queue itself and the synchronization mechanisms used to ensure correct access.

**3.1. Shared Data Structure**

* Queue (LinkedList<Integer>)
  + Holds the integer values in FIFO order.
  + Modified by multiple threads concurrently.

**3.2. Synchronization Mechanisms**

| **Approach** | **Purpose** |
| --- | --- |
| synchronized | Ensures mutual exclusion for method execution. |
| ReentrantLock | Provides finer control over locking and waiting conditions. |
| Semaphore | Controls access to prevent multiple simultaneous removals. |
| BlockingQueue | A built-in thread-safe queue that handles blocking and synchronization automatically. |

**3.3. Shared Data Implementation**

* The queue is protected by a ReentrantLock to prevent concurrent modifications.
* If removeFirst() is called on an empty queue, it waits until an element is added

**Summary**

| **Aspect** | **Thread-Safe Queue Implementation** |
| --- | --- |
| **Correctness Constraint** | Ensures FIFO ordering and correct element retrieval |
| **Waiting Condition** | removeFirst() waits if the queue is empty |
| **Shared State Used** | LinkedList<Integer> with ReentrantLock and Condition |

**Conclusion**

* A thread-safe queue was implemented using ReentrantLock and Condition.
* The queue maintains FIFO ordering and prevents race conditions.
* Multiple threads can add and remove elements concurrently without data corruption.
* If the queue is empty, removeFirst() waits until data is available, avoiding errors.
* Alternative approaches using synchronized or BlockingQueue can also be considered.

## **Assignment 3 (20 points)**

**1. Identify the Correctness Constraints of the Problem**

The Bounded Buffer must satisfy the following constraints:

1. **FIFO (First-In-First-Out) Order**
   * The remove() method must always remove the earliest inserted element (first in the queue).
   * The add() method must always insert a new element at the end of the queue.
2. **Thread Safety**
   * Multiple producer and consumer threads should be able to concurrently modify the buffer without causing race conditions.
   * The buffer must prevent data corruption when accessed by multiple threads.
3. **Blocking Behavior**
   * If the buffer is full, any thread calling add() must wait until space is available.
   * If the buffer is empty, any thread calling remove() must wait until an element is available.
4. **Avoiding Deadlocks & Starvation**
   * Threads waiting to add or remove elements must be notified promptly when the buffer state changes.
   * There should be fairness so that all threads get a chance to proceed.

**2. Specify the Conditions That Each Method Must Wait For**

| **Method** | **Waiting Condition** | | **Synchronization Mechanism** |
| --- | --- | --- | --- |
| add(int x) | Waits if the buffer is full | ReentrantLock + Condition | |
| remove() | Waits if the buffer is empty | ReentrantLock + Condition | |

**Synchronization Approach**

**Using ReentrantLock and Condition Variables**

* + A thread must acquire a lock (lock.lock()) before modifying the buffer.
  + If the buffer is full, add() must wait using notFull.await().
  + If the buffer is empty, remove() must wait using notEmpty.await().
  + When an element is added, notEmpty.signal() is called to wake up waiting consumers.
  + When an element is removed, notFull.signal() is called to wake up waiting producers.

**3. Write Down the Shared State That You Will Use to Check These Conditions**

* Buffer (int[] buffer) → Fixed-size array to store integers.
* Size (int capacity) → Maximum number of elements allowed in the buffer.
* Current Count (int count) → Keeps track of the number of elements currently in the buffer.
* Index Pointers (int head, tail) → Track the front and rear positions of the buffer (circular queue).
* Synchronization Mechanisms:
  + ReentrantLock → Ensures mutual exclusion for add() and remove().
  + Condition Variables (notEmpty, notFull) → Manage waiting threads when the buffer is full or empty.

## **Assignment 4 (20 points)**

**1. Identify the Correctness Constraints of the Problem**

The Barrier class must ensure the following constraints:

1. All Threads Must Wait at the Barrier
   * A thread calling await() must block until the required number of threads arrive.
   * No thread should proceed before all required threads have called await().
2. Synchronization Must Be Properly Handled
   * Threads must not proceed independently; they should wait for others.
   * The barrier should reset after all threads pass the barrier to allow reuse.
3. Avoid Deadlocks and Ensure Fairness
   * No thread should be stuck indefinitely due to race conditions.
   * All waiting threads must be released simultaneously when the barrier is tripped.

**2. Specify the Conditions That Each Method Must Wait For**

| **Method** | **Waiting Condition** | **Synchronization Mechanism** |
| --- | --- | --- |
| await() | Waits until the required number of threads have called await() | ReentrantLock + Condition |

**Synchronization Approach**

Using ReentrantLock and Condition Variables

* + A thread must acquire a lock (lock.lock()) before modifying the barrier counter.
  + If the required number of threads has not arrived, calling await() will wait using condition.await().
  + When the last required thread arrives, it notifies all waiting threads (condition.signalAll()) to continue execution.
  + The barrier resets automatically for reuse.

**3. Write Down the Shared State That You Will Use to Check These Conditions**

* Thread Counter (int count) → Tracks how many threads have arrived at the barrier.
* Total Parties (int parties) → Defines the required number of threads to trip the barrier.
* Synchronization Mechanisms:
  + ReentrantLock → Ensures mutual exclusion for updating count.
  + Condition Variable (barrierCondition) → Manages threads waiting at the barrier.

## **Assignment 5 (20 points)**

**1. Identify the Correctness Constraints of the Problem**

The **Old Bridge Monitor** must ensure the following constraints:

1. **Maximum 3 Cars on the Bridge**
   * The bridge must never have more than 3 cars at any time.
2. **Cars Must Move in the Same Direction**
   * At any moment, all cars on the bridge must be going in the same direction (either 0 or 1).
3. **Synchronization & Mutual Exclusion**
   * When a car arrives, it must wait if:
     + The bridge has 3 cars already.
     + The bridge has cars going in the opposite direction.
   * When a car exits, it must notify waiting cars that they may proceed.
4. **Fair Usage, but No Starvation Handling**
   * The solution does not need to prevent one direction from being blocked for too long.
   * The only requirement is that cars must always be on the bridge when possible.

**2. Specify the Conditions That Each Method Must Wait For**

| **Method** | **Waiting Condition** | **Synchronization Mechanism** |
| --- | --- | --- |
| arriveBridge(int direction) | Waits if the bridge is full or if cars are going in the opposite direction | ReentrantLock + Condition |
| exitBridge() | Signals waiting cars if space is available or direction can change | ReentrantLock + Condition |

**Synchronization Approach**

**Using ReentrantLock and Condition Variables**

* + A car must acquire a lock (lock.lock()) before modifying the bridge state.
  + If the bridge is full or has cars going the other way, arriveBridge() must wait using condition.await().
  + When a car exits, it notifies waiting cars (condition.signalAll()) to check if they can cross.
  + The direction only changes when the bridge is empty.

**3. Write Down the Shared State That You Will Use to Check These Conditions**

* Current Car Count (int count) → Tracks how many cars are currently on the bridge.
* Current Direction (int currentDirection) → Tracks which direction the cars on the bridge are going.
* Synchronization Mechanisms:
  + ReentrantLock → Ensures mutual exclusion for updating count and currentDirection.
  + Condition Variable (bridgeCondition) → Manages waiting cars.