DAY 23:

ASSIGNMENTS:

Task 3: Synchronization and Inter-thread Communication

Implement a producer-consumer problem using wait() and notify() methods to handle the correct processing sequence between threads.

```
import java.util.LinkedList;
import java.util.Queue;

class SharedBuffer {
    private final Queue<Integer> queue = new LinkedList<>();
    private final int capacity;

    public SharedBuffer(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void produce(int value) throws InterruptedException {
        while (queue.size() == capacity) {
            wait(); // Wait if the buffer is full
```

```
        }
        queue.add(value);
        System.out.println("Produced: " + value);
        notify(); // Notify consumer that they can consume
    }

    public synchronized int consume() throws InterruptedException {
        while (queue.isEmpty()) {
            wait(); // Wait if the buffer is empty
        }
        int value = queue.remove();
        System.out.println("Consumed: " + value);
        notify(); // Notify producer that they can produce
        return value;
    }
}

class Producer implements Runnable {
    private final SharedBuffer buffer;

    public Producer(SharedBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
```

```
        for (int i = 0; i < 10; i++) {
            try {
                buffer.produce(i);
                Thread.sleep(300); // Simulate time taken to produce an item
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class Consumer implements Runnable {
    private final SharedBuffer buffer;

    public Consumer(SharedBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.consume();
                Thread.sleep(600); // Simulate time taken to consume an item
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}

public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer(5);

        Thread producerThread = new Thread(new Producer(buffer));
        Thread consumerThread = new Thread(new Consumer(buffer));

        producerThread.start();
        consumerThread.start();

        try {
            producerThread.join();
            consumerThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Producer and Consumer have finished their tasks.");
    }
}
```

Explanation:

1. *SharedBuffer Class*:

   - A shared buffer implemented using a Queue with a fixed capacity.

   - The produce method adds items to the buffer. If the buffer is full, the producer thread waits using wait().

   - The consume method removes items from the buffer. If the buffer is empty, the consumer thread waits using wait().

   - Both methods are synchronized to ensure mutual exclusion, and use notify() to wake up waiting threads.

2. *Producer Class*:

   - Implements the Runnable interface.

   - Produces items and adds them to the buffer. It simulates the time taken to produce an item using Thread.sleep(300).

3. *Consumer Class*:

   - Implements the Runnable interface.

   - Consumes items from the buffer. It simulates the time taken to consume an item using Thread.sleep(600).

4. *ProducerConsumerExample Class*:

   - Creates instances of the SharedBuffer, Producer, and Consumer classes.

   - Starts the producer and consumer threads.

   - Waits for both threads to finish using join().

Running the Program:

When you run this program, you will see the producer and consumer threads producing and consuming items in sequence. The wait() and notify() methods ensure that the producer waits if the buffer is full and the consumer waits if the buffer is empty, maintaining the correct processing sequence. This example further emphasizes the importance of synchronization in managing shared resources between threads.