4. Process Synchronization Using Semaphores

Aim:

To implement semaphores to solve the producer-consumer problem with bounded buffer.

Program:

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.Semaphore;

/**
 * A Java program to demonstrate the solution to the
 * producer consumer problem, using a bounded buffer
 * and semaphores.
 *
 * @author jimil
 */
public class ProducerConsumerSemaphore {

      private static Queue<Integer> buffer = new LinkedList<Integer>();
      private static List<Thread> threads = new ArrayList<Thread>();
      private final int MAX_BUFFER_SIZE = 3;
      /**
       * new Semaphore(1) creates a new semaphore variable
       * with maximum allowable permits = 1.
       * (Binary Semaphore)
       */
      public static Semaphore mutex = new Semaphore(1);

      /**
       * Driver method for program
       *
```

```
	 * @param args
	 */
	public static void main(String[] args) throws Exception {
		for(int i = 0; i < 10; i++) {
			int rand = new Random().nextInt(1000) + 2;
			if(rand%2 == 0) {
				Producer producer = new Producer(new
StringBuffer().append(i).toString());
				threads.add(producer);
				producer.start();
			}
			else {
				Consumer consumer = new Consumer(new
StringBuffer().append(i).toString());
				threads.add(consumer);
				consumer.start();
			}
			Thread.sleep(1000);
		}
		for(Thread thread: threads) {
			thread.join();
		}
	}

	/**
	 * Utility function that is used by the producer to
	 * add a new item to the buffer.
	 *
	 * @param next_produced
	 * @throws BufferFilledException
	 * @return
	 */
	public boolean addToBuffer(int next_produced) {
		if(this.buffer.size() >= MAX_BUFFER_SIZE) {
			System.out.println("Buffer has exceeded it's maximum
limit. Cannot Produce!");
			return false;
		} else {
			this.buffer.add(next_produced);
			return true;
		}
	}
```

```
	/**
	 * Utility function that is used by the consumer to
	 * consume produced items from the buffer.
	 *
	 * @return
	 * @throws BufferEmptyException
	 */
	public Integer removeFromBuffer() {
		if(this.buffer.isEmpty()) {
			System.out.println("Buffer is currently empty. Cannot
Consume!");
		} else {
			return this.buffer.remove();
		}
		return 0;
	}
}

/**
 * Producer is a process that produces an item
 * and enqueues it in the bounded buffer.
 *
 * @author jimil
 */
class Producer extends Thread {

	ProducerConsumerSemaphore obj = new ProducerConsumerSemaphore();

	/**
	 * Parameterized constructor to set thread's name.
	 *
	 * @param id
	 */
	public Producer(String id) {
		super.setName(id);
	}

	/**
	 * Generates a random number and adds to buffer.
	 *
	 * @throws Exception
```

```
	 */
	public void produce() throws Exception {
		try {
			// Generate a random number and add to buffer
			int next_produced = new Random().nextInt(1000) + 1;
			boolean result = obj.addToBuffer(next_produced);
			if(result) {
				System.out.println("Producer " + this.getName() + "
generated: " + next_produced);
			}
		} catch (Exception e) {
			throw new Exception(e);
		}
	}

	/**
	 * Overriden implementation for default method
	 */
	@Override
	public void run() {
		try {
			// Wait for permit
			ProducerConsumerSemaphore.mutex.acquire();
			// Critical Section
			produce();
			// Release the permit
			ProducerConsumerSemaphore.mutex.release();
		} catch(Exception e) {
			e.printStackTrace();
		}
	}

}

/**
 * Consumer is a process that consumes an item
 * by dequeuing it from the bounded buffer.
 *
 * @author jimil
 */
class Consumer extends Thread {
```

```
	ProducerConsumerSemaphore obj = new ProducerConsumerSemaphore();

	public Consumer(String id) {
		super.setName(id);
	}

	/**
	 * Reads the latest item from buffer.
	 *
	 * @throws Exception
	 */
	public void consume() throws Exception {
		try {
			// Generate a random number and add to buffer
			int next_consumed = obj.removeFromBuffer();
			if(next_consumed > 0) {
				System.out.println("Consumer " + this.getName() + "
consumed: " + next_consumed);
			}
		} catch (Exception e) {
			throw new Exception(e);
		}
	}

	/**
	 * Overriden implementation for default method
	 */
	@Override
	public void run() {
		try {
			// Wait for permit
			ProducerConsumerSemaphore.mutex.acquire();
			// Critical Section
			consume();
			// Release the permit
			ProducerConsumerSemaphore.mutex.release();
		} catch(Exception e) {
			e.printStackTrace();
		}
	}

}
```

Output:

```
jimil@jimil-Lenovo-G50-80:~/Documents/SPIT/Sem 4/OS/Pracs/Lab 4$ javac
ProducerConsumerSemaphore.java
jimil@jimil-Lenovo-G50-80:~/Documents/SPIT/Sem 4/OS/Pracs/Lab 4$ java
ProducerConsumerSemaphore
Buffer is currently empty. Cannot Consume!
Producer 1 generated: 448
Consumer 2 consumed: 448
Producer 3 generated: 681
Consumer 4 consumed: 681
Producer 5 generated: 297
Producer 6 generated: 388
Producer 7 generated: 107
Buffer has exceeded it’s maximum limit. Cannot Produce!
Consumer 9 consumed: 297
```

Conclusion:

Thus, I used the concept of semaphores to implement a solution to the producer-consumer problem using a bounded buffer.