Producer-Consumer solution using Semaphores in Java | Set 2

Prerequisites – Semaphore in Java, Inter Process Communication, Producer Consumer Problem using Semaphores | Set 1

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem : To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution : The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

In the post Producer-Consumer solution using threads in Java, we have discussed above solution by using inter-thread communication(wait(), notify(), sleep()). In this post, we will use Semaphores to implement the same.

The below solution consists of four classes:

1. **Q** : the queue that you're trying to synchronize
2. **Producer :** the threaded object that is producing queue entries
3. **Consumer :** the threaded object that is consuming queue entries
4. **PC :** the driver class that creates the single Q, Producer, and Consumer.

```
// Java implementation of a producer and consumer
// that use semaphores to control synchronization.

import java.util.concurrent.Semaphore;

class Q
{
    // an item
    int item;

    // semCon initialized with 0 permits
    // to ensure put() executes first
    static Semaphore semCon = new Semaphore(0);

    static Semaphore semProd = new Semaphore(1);
```

```
    // to get an item from buffer
    void get()
    {
        try {
            // Before consumer can consume an item,
            // it must acquire a permit from semCon
            semCon.acquire();
        }
        catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

        // consumer consuming an item
        System.out.println("Consumer consumed item : " + item);

        // After consumer consumes the item,
        // it releases semProd to notify producer
        semProd.release();
    }

    // to put an item in buffer
    void put(int item)
    {
        try {
            // Before producer can produce an item,
            // it must acquire a permit from semProd
            semProd.acquire();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

        // producer producing an item
        this.item = item;

        System.out.println("Producer produced item : " + item);

        // After producer produces the item,
        // it releases semCon to notify consumer
        semCon.release();
    }
}

// Producer class
class Producer implements Runnable
{
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        for(int i=0; i < 5; i++)
            // producer put items
            q.put(i);
    }
}

// Consumer class
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q){
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run()
    {
        for(int i=0; i < 5; i++)
            // consumer get items
            q.get();
    }
}

// Driver class
class PC
{
    public static void main(String args[])
    {
        // creating buffer queue
        Q q = new Q();

        // starting consumer thread
        new Consumer(q);
```

```
        // starting producer thread
        new Producer(q);
    }
}
```

Output:

```
Producer produced item : 0
Consumer consumed item : 0
Producer produced item : 1
Consumer consumed item : 1
Producer produced item : 2
Consumer consumed item : 2
Producer produced item : 3
Consumer consumed item : 3
Producer produced item : 4
Consumer consumed item : 4
```

Explanation : As you can see, the calls to **put()** and **get( )** are synchronized, i.e. each call to put() is followed by a call to get( ) and no items are missed. Without the semaphores, multiple calls to put() would have occurred without matching calls to get(), resulting in items being missed. (To prove this, remove the semaphore code and observe the results.)

The sequencing of put() and get() calls is handled by two semaphores: semProd and **semCon**.

- Before put( ) can produce an item, it must acquire a permit from semProd. After it has produce the item, it releases semCon.
- Before get( ) can consume an item, it must acquire a permit from semCon. After it consumes the item, it releases semProd.
- This “give and take” mechanism ensures that each call to put( ) must be followed by a call to get( ).
- Also notice that semCon is initialized with no available permits. This ensures that put( ) executes first. The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

Producer-Consumer solution using threads in Java
Semaphores in Process Synchronization
Longest prefix matching - A Trie based solution in Java
Dining Philosopher Problem Using Semaphores
Producer Consumer Problem using Semaphores | Set 1
Reader-Writers solution using Monitors
Dining-Philosophers Solution Using Monitors
Classical problems of Synchronization with Semaphore Solution
Readers-Writers Problem | Set 1 (Introduction and Readers Preference Solution)
Java.util.LinkedList.poll(), pollFirst(), pollLast() with examples in Java
Java.util.concurrent.Phaser class in Java with Examples
Java.util.function.IntPredicate interface in Java with Examples
Java.util.concurrent.RecursiveAction class in Java with Examples