Custom Semaphore used for implementing Producer Consumer pattern in java

You are here : *Home* / *Core Java Tutorials* / *Thread Concurrency Tutorial in java*

In previous thread concurrency tutorial we learned how to implement your own/custom **Semaphore** in java.
Now we will learn ***Application** of custom/own Semaphore in real world (for solving Producer Consumer problem in java).*

Contents of page:

In previous posts we learned how to use Semaphores in java, and also implemented custom Semaphore. Now, let's use Semaphore for implementing Producer Consumer pattern.

*1) **Logic** behind using **Custom Semaphore** for implementing **Producer Consume** pattern in java >*

Semaphore on producer is created with permit =1. So, that **producer can get the permit to produce**.
Semaphore on consumer is created with permit =0. So, that **consumer could wait for permit to consume**. [because initially producer hasn't produced any product]

Producer gets permit by calling **semaphoreProducer.acquire()** and **starts producing**, **after producing** it calls **semaphoreConsumer.release()**. So, that **consumer could get the to consume**.

```
semaphoreProducer.acquire();
System.out.println("Produced : "+i);
semaphoreConsumer.release();
```

Consumer gets permit by calling **semaphoreConsumer.acquire()** and **starts consuming**, **after consuming** it calls **semaphoreProducer.release()**. So, that **producer could get t permit to produce**.

```
semaphoreConsumer.acquire();
System.out.println("Consumed : "+i);
semaphoreProducer.release();
```

*2) **Program** to demonstrate usage of **Custom Semaphore** for implementing Produc Consumer pattern in java >*

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
/**
* @author AnkitMittal
* Copyright (c), AnkitMittal .
* All Contents are copyrighted and must not be reproduced in any form.
* A semaphore controls access to a shared resource by using permits.
    - If permits are greater than zero, then semaphore
      allow access to shared resource.
    - If permits are zero or less than zero, then semaphore
      does not allow access to shared resource.

*/
class SemaphoreCustom{

    private int permits;

    /** permits is the initial number of permits available.
           This value can be negative, in which case releases must occur
           before any acquires will be granted, permits is number of threads
           that can access shared resource at a time.
           If permits is 1, then only one threads that can access shared
           resource at a time.
    */
    public SemaphoreCustom(int permits) {
           this.permits=permits;
    }

    /**Acquires a permit if one is available and decrements the
       number of available permits by 1.
           If no permit is available then the current thread waits
           until one of the following things happen >
            >some other thread calls release() method on this semaphore or,
            >some other thread interrupts the current thread.
    */
    public synchronized void acquire() throws InterruptedException {
           //Acquires a permit, if permits is greater than 0 decrements
           //the number of available permits by 1.
           if(permits > 0){
                  permits--;
           }
           //permit is not available wait, when thread
           //is notified it decrements the permits by 1
           else{
                  this.wait();
                  permits--;
           }
    }

    /** Releases a permit and increases the number of available permits by 1.
           For releasing lock by calling release() method it’s not mandatory
           that thread must have acquired permit by calling acquire() method.
    */
    public synchronized void release() {
           //increases the number of available permits by 1.
           permits++;

           //If permits are greater than 0, notify waiting threads.
           if(permits > 0)
                  this.notify();
    }
}


/**
 * Main class, for testing SemaphoreCustom
 */
public class SemaphoreCustomConsumerProducer{

    public static void main(String[] args) {

           SemaphoreCustom semaphoreProducer=new SemaphoreCustom(1);
           SemaphoreCustom semaphoreConsumer=new SemaphoreCustom(0);
           System.out.println("semaphoreProducer permit=1 | semaphoreConsumer permit=0");

       Producer producer=new Producer(semaphoreProducer,semaphoreConsumer);
       Consumer consumer=new Consumer(semaphoreConsumer,semaphoreProducer);

        Thread producerThread = new Thread(producer, "ProducerThread");
        Thread consumerThread = new Thread(consumer, "ConsumerThread");

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

    }
}




/**
 * Producer Class.
 */
class Producer implements Runnable{

    SemaphoreCustom semaphoreProducer;
    SemaphoreCustom semaphoreConsumer;


    public Producer(SemaphoreCustom semaphoreProducer,SemaphoreCustom semaphoreConsumer) {
           this.semaphoreProducer=semaphoreProducer;
           this.semaphoreConsumer=semaphoreConsumer;
    }

    public void run() {
           for(int i=1;i<=5;i++){
                  try {
                        semaphoreProducer.acquire();
                        System.out.println("Produced : "+i);
                        semaphoreConsumer.release();

                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
           }
    }
}

/**
```

```
 * Consumer Class.
 */
class Consumer implements Runnable{

    SemaphoreCustom semaphoreConsumer;
    SemaphoreCustom semaphoreProducer;

    public Consumer(SemaphoreCustom semaphoreConsumer,SemaphoreCustom semaphoreProducer) {
           this.semaphoreConsumer=semaphoreConsumer;
           this.semaphoreProducer=semaphoreProducer;
    }

    public void run() {

           for(int i=1;i<=5;i++){
                  try {
                        semaphoreConsumer.acquire();
                        System.out.println("Consumed : "+i);
                        semaphoreProducer.release();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
           }
    }

}


/*OUTPUT

semaphoreProducer permit=1 | semaphoreConsumer permit=0
Produced : 1
Consumed : 1
Produced : 2
Consumed : 2
Produced : 3
Consumed : 3
Produced : 4
Consumed : 4
Produced : 5
Consumed : 5

*/
```

*Let's discuss output in detail, to get better understanding of how we have used **Cus** **Semaphore** for implementing Producer Consumer pattern in java >*

Note : (I have mentioned output in `green` text and it's explanation is given in line immediately followed by it)

```
semaphoreProducer permit=1 | semaphoreConsumer permit=0
```

semaphoreProducer created with permit=1. So, that producer can get the permit to produce |
semaphoreConsumer created with permit=0. So, that consumer could wait for permit to consume.

semaphoreProducer.acquire() is called, Producer has got the permit and it can produce [Now, semaphoreProducer permit=0]
`Produced : 1` [as producer has got permit, it is producing]
semaphoreConsumer.release() is called, `Permit has been released on` semaphoreConsumer means consumer can consume `[Now,` semaphoreConsumer permit=1]

semaphoreConsumer.acquire() is called, `Consumere has got the permit and it can consume [Now,` semaphoreConsumer permit=0]
`Consumed : 1` [as consumer has got permit, it is consuming]
semaphoreProducer.release() is called, `Permit has been released on` semaphoreProducer means producer can produce `[Now,` semaphoreProducer permit=1]

```
Produced : 2
Consumed : 2
Produced : 3
Consumed : 3
Produced : 4
Consumed : 4
Produced : 5
Consumed : 5
```