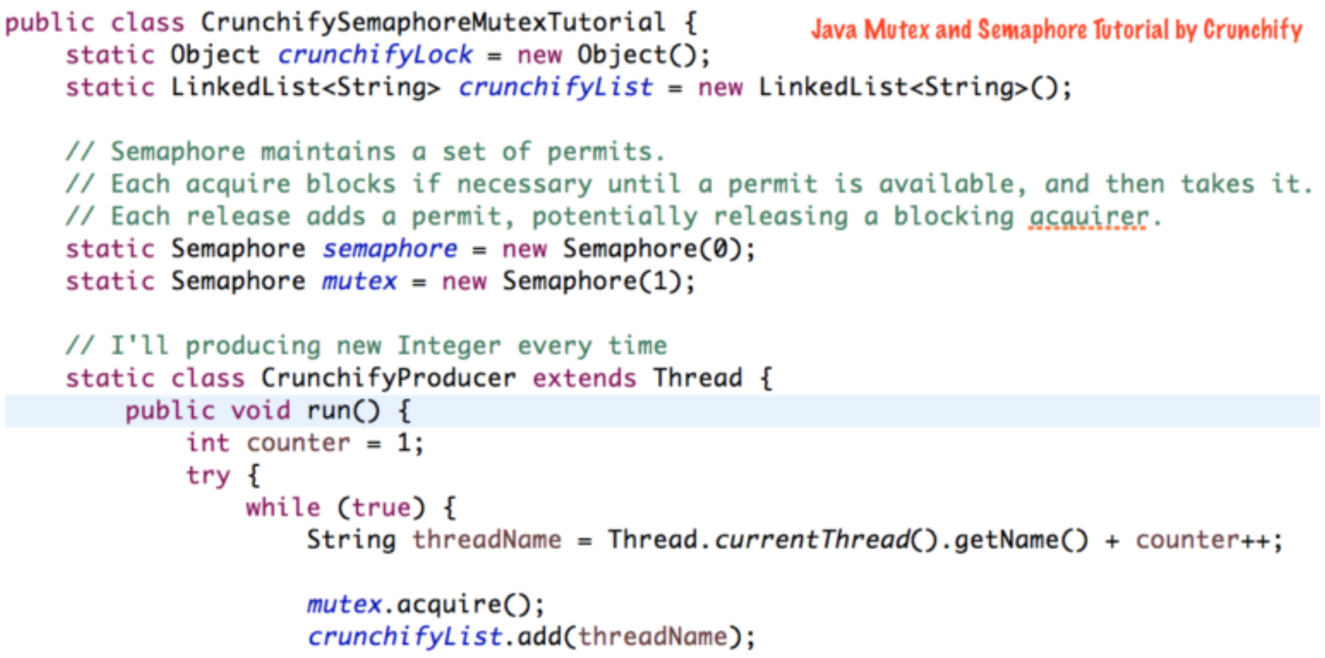
Java Code Spring MVC Blogging Tomcat Tips Tutorials+ WordPress Deals Advertise Contact

BLOG POST

Crunchify ▸ Core Java and J2EE Tutorials and Tips ▸ What is Java Semaphore and Mutex – Java Concurrency Mu ...

What is Java Semaphore and Mutex – Java Concurrency MultiThread explained with Example

Last Updated on June 6th, 2016 by App Shah 2 Comments http://crunchify.me/1VIC6a



```
public class CrunchifySemaphoreMutexTutorial {
    static Object crunchifyLock = new Object();
    static LinkedList<String> crunchifyList = new LinkedList<String>();

    // Semaphore maintains a set of permits.
    // Each acquire blocks if necessary until a permit is available, and then takes it.
    // Each release adds a permit, potentially releasing a blocking acquirer.
    static Semaphore semaphore = new Semaphore(0);
    static Semaphore mutex = new Semaphore(1);

    // I'll producing new Integer every time
    static class CrunchifyProducer extends Thread {
        public void run() {
            int counter = 1;
            try {
                while (true) {
                    String threadName = Thread.currentThread().getName() + counter++;

                    mutex.acquire();
                    crunchifyList.add(threadName);
```

Java Concurrency is a very wide topic. There are hundreds of tutorials and examples available for use to use. Some time back I've written few tutorials on Run Multiple Threads Concurrently in Java and different types of Synchronized Blocks.

In this tutorial we will go over:

Let’s get started

`Let's keep this in mind` while reading below explanation:

- Take an example of Shopper and Customer
- Shopper is borrowing Laptops
- Customer can come and use Laptop – customer need a key to use a Laptop
- After use – customer can return Laptop to Shopper

What is Mutex (Just 1 thread):

Shopper has a key to a Laptop. One customer can have the key – borrow a Laptop – at the time. When task finishes, the Shopper gives (frees) the key to the next customer in the queue.

`Official Definition`: “Mutexes are typically used to serialise access to a section of `re-entrant code` that `cannot be executed concurrently` by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.”

In other words: `Mutex = Mutually Exclusive Semaphore`

What is Semaphore (N specified threads):

Let’s say now Shopper has 3 identical Laptops and 3 identical keys. Semaphore is the number of `free identical Laptop keys`. The semaphore count – the count of keys – is set to 3 at beginning (all three Laptops are free), then the count value is decremented as Customer are coming in. If all Laptops are in use, i.e. there are no free keys left for Laptop, the semaphore count is 0. Now, when any of the customer returns the Laptop, semaphore is increased to 1 (one free key), and given to the next customer in the queue.

Another must read: Lazy Creation of Singleton ThreadSafe Instance

Example-1: (Explanation below)

CrunchifySemaphoreMutexTutorial.java Java

```java
package crunchify.com.tutorial;

import java.util.LinkedList;
import java.util.concurrent.Semaphore;

/**
 * @author Crunchify.com
 *
 */

public class CrunchifySemaphoreMutexTutorial {
    static Object crunchifyLock = new Object();
    static LinkedList<String> crunchifyList = new LinkedList<String>();

    // Semaphore maintains a set of permits.
    // Each acquire blocks if necessary until a permit is available, and then takes
    // Each release adds a permit, potentially releasing a blocking acquirer.
    static Semaphore semaphore = new Semaphore(0);
    static Semaphore mutex = new Semaphore(1);

    // I'll producing new Integer every time
    static class CrunchifyProducer extends Thread {
        public void run() {

            int counter = 1;
            try {
                while (true) {
                    String threadName = Thread.currentThread().getName() + counter+

                    mutex.acquire();
                    crunchifyList.add(threadName);
                    System.out.println("Producer is prdoucing new value: " + thread
                    mutex.release();

                    // release lock
                    semaphore.release();
    }

    // I'll be consuming Integer every stime
    static class CrunchifyConsumer extends Thread {
        String consumerName;

        public CrunchifyConsumer(String name) {
            this.consumerName = name;
        }

        public void run() {
            try {

                while (true) {

                    // acquire lock. Acquires the given number of permits from this
                    // available
                    // process stops here until producer releases the lock
                    semaphore.acquire();

                    // Acquires a permit from this semaphore, blocking until one is
                    mutex.acquire();
                    String result = "";
                    for (String value : crunchifyList) {
                        result = value + ",";
                    }
                    System.out.println(consumerName + " consumes value: " + result
                            + crunchifyList.size() + "\n");
                    mutex.release();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        new CrunchifyProducer().start();
        new CrunchifyConsumer("Crunchify").start();
        new CrunchifyConsumer("Google").start();
        new CrunchifyConsumer("Yahoo").start();
    }
}
```

Upon signalling a added data, one of the consumers will be woken and it will know it can read a crunchifyList Object. It can read a list, then go back to trying to acquire on the semaphore. If in that time the producer has written another packet it has signalled again and either of the consumers will then go on to read another packet and so on…

In other words:

```
CrunchifyProducer: Add an object o List - Semaphore.release(1)

CrunchifyConsumer x N) - Semaphore.acquire(1) - Read an object from List
```

Result:

```
Result
Producer is prdoucing new value: Thread-01
Crunchify consumes value: Thread-01,crunchifyList.size(): 1

Producer is prdoucing new value: Thread-02
Google consumes value: Thread-02,crunchifyList.size(): 2

Producer is prdoucing new value: Thread-03
Yahoo consumes value: Thread-03,crunchifyList.size(): 3

Producer is prdoucing new value: Thread-04
Crunchify consumes value: Thread-04,crunchifyList.size(): 4


.....
.....
.....
```

How to prevent race condition:

`What if you have multiple Consumers?` In above Java Tutorial The consumers (not the producer) should lock the buffer when reading the packet (but not when acquiring the

CrunchifyJavaSemaphoreTutorial.java Java

```java
package crunchify.com.tutorial;

import java.util.concurrent.Semaphore;

/**
 * @author Crunchify.com
 *
 */

public class CrunchifyJavaSemaphoreTutorial {
    private static final int MAX_CONCURRENT_THREADS = 2;
    private final Semaphore crunchifyAdminLOCK = new Semaphore(MAX_CONCURRENT_THREA

    public void crunchifyStartTest() {
        for (int i = 0; i < 10; i++) {
            CrunchifyPerson person = new CrunchifyPerson();
            person.start();
        }
    }

    public class CrunchifyPerson extends Thread {
        @Override
        public void run() {
            try {

                // Acquire Lock
                crunchifyAdminLOCK.acquire();
            } catch (InterruptedException e) {
                System.out.println("received InterruptedException");
                return;
            }
            System.out.println("Thread " + this.getId() + " start using Crunchify's
            try {
                sleep(1000);
            } catch (Exception e) {

            } finally {

                // Release Lock
                crunchifyAdminLOCK.release();
            }
            System.out.println("Thread " + this.getId() + " stops using Crunchify's
    }
}
```

Result:

Result

```
Thread 11 start using Crunchify's car - Acquire()
Thread 10 start using Crunchify's car - Acquire()
Thread 10 stops using Crunchify's car -  Release()

Thread 12 start using Crunchify's car - Acquire()
Thread 13 start using Crunchify's car - Acquire()
Thread 11 stops using Crunchify's car -  Release()

Thread 13 stops using Crunchify's car -  Release()

Thread 15 start using Crunchify's car - Acquire()
Thread 14 start using Crunchify's car - Acquire()
Thread 12 stops using Crunchify's car -  Release()

Thread 14 stops using Crunchify's car -  Release()

Thread 16 start using Crunchify's car - Acquire()
Thread 15 stops using Crunchify's car -  Release()

Thread 17 start using Crunchify's car - Acquire()
Thread 17 stops using Crunchify's car -  Release()

Thread 18 start using Crunchify's car - Acquire()
Thread 19 start using Crunchify's car - Acquire()
Thread 16 stops using Crunchify's car -  Release()

Thread 18 stops using Crunchify's car -  Release()

Thread 19 stops using Crunchify's car -  Release()
```

Buffer Follow

TOP DEALS