Java Concurrency Tutorial – Semaphores

javacodegeeks.com Byron Kiourtzoglou September 15th, 2011 view original

This is the first part in a series that we're going to be doing on Java concurrency. Specifically, we are going to dive into the concurrency tools built into Java 1.5 and beyond. We're going to assume you have a basic understanding of synchronization and volatile keywords.
The first post will cover semaphores - specifically **counting semaphores**. Semaphores are an often misunderstood and underused tool for restricting access to resources. They are ignored for other ways of controlling access to resources. But semaphores give us a toolset that goes beyond what normal synchronization and other tools can give us.

So what is semaphore? The simplest way to think of a semaphore is to consider it an abstraction that allows n units to be acquired, and offers acquire and release mechanisms. It safely allows us to ensure that **only n processes can access a certain resource at a given time**.

That's all well and good, but what purpose would this serve? Well, here is one example that will help explain its uses. It uses the nicely designed Semaphore class introduced in 1.5, located in the java.util.concurrent package.

Limiting connections

Perhaps we have a process that downloads resources for us periodically via HTTP. We don't want to spam any of the hosts and at the same time, we want to limit how many connections we are making so we don't exhaust the limited file handles or outbound connections we are permitted. A simple way to do this would be with a semaphore:

```
public class ConnectionLimiter {
   private final Semaphore semaphore;
```

```
    private ConnectionLimiter(int maxConcurrentRequests) {
        semaphore = new Semaphore(maxConcurrentRequests);
    }

    public URLConnection acquire(URL url) throws
InterruptedException,

IOException {
        semaphore.acquire();
        return url.openConnection();
    }

    public void release(URLConnection conn) {
        try {
            /*
            * ... clean up here
            */
        } finally {
            semaphore.release();
        }
    }
}
```

This is a nice elegant solution to a problem of limited resources. The call to **acquire()** will block until permits are available. The beauty of the semaphore is that it hides all the complexity of managing access control, counting permits and, of course, getting the thread-safety right.
Dangers

As with most methods of locking or synchronization, there are some potential issues.

The number one thing to remember is, **always release what you acquire**. This is done by using try..finally constructs.

There are other less obvious problems that can befall you when using semaphores. The following class shows a deadlock that only the luckiest of you will avoid. You'll notice that the two threads which acquire the two semaphore permits do so in opposite order. (try..finally is left out for the sake of brevity).

```
public static void main(String[] args) throws Exception {
   Semaphore s1 = new Semaphore(1);
   Semaphore s2 = new Semaphore(1);

   Thread t = new Thread(new DoubleResourceGrabber(s1,
s2));
   // now reverse them ... here comes trouble!
   Thread t2 = new Thread(new DoubleResourceGrabber(s2,
s1));

   t.start();
   t2.start();

   t.join();
   t2.join();
   System.out.println("We got lucky!");
}

private static class DoubleResourceGrabber implements
Runnable {
   private Semaphore first;
   private Semaphore second;

   public DoubleResourceGrabber(Semaphore s1, Semaphore
s2) {
       first = s1;
       second = s2;
   }

   public void run() {
```

```
            try {
                Thread t = Thread.currentThread();

                first.acquire();
                System.out.println(t + " acquired " + first);

                Thread.sleep(200); // demonstrate deadlock

                second.acquire();
                System.out.println(t + " acquired " + second);

                second.release();
                System.out.println(t + " released " + second);

                first.release();
                System.out.println(t + " released " + first);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
```

If you run this, you will more than likely have a hung process. Issues of **lock ordering** apply to semaphores as much as regular mutexes or synchronization in Java. In some cases, timeouts (see note on tryAcquire() later in the article) can be used to prevent deadlocks from causing a process to hang up, but typically a deadlock is a symptom of a logic error which can be avoided. If you're unfamiliar with deadlocks, I recommend you read up on them. Wikipedia has a decent article on deadlocks which applies to all languages equally.
The main things that you should be careful of when using semaphores (including binary semaphores, i.e. mutexes) are:

- Not releasing after acquire (either missing release call or an exception is thrown and there is no finally block)
- Long held semaphores, causing thread starvation

- Deadlocks (as seen above)

Useful Tricks with Semaphores

One interesting property of Semaphores in Java is that **release doesn't have to be called by the same thread as acquire**. This means you could have a thread limiter that pools or creates threads based on a semaphore by calling acquire(). Then, the running thread could release its own semaphore permit when it completes. This is a useful property that we don't have with normal mutexes in Java.

Another trick is to **increase the number of permits** at runtime. Contrary to what you might guess, the number of permits in a semaphore isn't fixed, and a call to **release()** will always increment the number of permits, even if no corresponding **acquire()** call was made. Note that this can also result in bugs if you are incorrectly calling **release()** when no **acquire()** was made.

Finally, there are a few useful methods to be familiar with in Java's Semaphore. The method **acquireInterruptibly()** will acquire a resource, reattempting if it is interrupted. This means no outside handling of InterruptedException. The method **tryAcquire()** allows us to limit how long we will wait for a permit – we can either return immediately if there is no permit to obtain, or wait a specified timeout. If you somehow have known deadlocks that you can't fix easily or track down, you could help prevent locking up processes by using **tryAcquire()** with suitable timeouts.

Uses

What are some possible uses for counting semaphores? The following come to mind:

- Limiting concurrent access to disk (this can kill performance due to competing disk seeks)
- Thread creation limiting
- JDBC connection pooling / limiting
- Network connection throttling
- Throttling CPU or memory intensive tasks

Of course, a semaphore is a pretty low-level building block for access control and synchronization. Java

provides us a wealth of concurrency mechanisms and strategies which were introduced in Java 1.5 and beyond. In the coming posts, we will be covering some of the more abstract methods of managing concurrency including Executors, BlockingQueues, Barriers, Futures and also some of the new concurrent Collection classes.
What uses have you found for semaphores? Let us know by leaving a comment – we love talking software.

Reference: Java Concurrency Part 1 – Semaphores from our JCG partners at the Carfey Software blog.