Implementing Semaphore

Learn how to design and implement a simple semaphore class in Java.

Problem

Java does provide its own implementation of Semaphore, however, Java's semaphore is initialized with an initial number of permits, rather than the maximum possible permits and the developer is expected to take care of always releasing the intended number of maximum permits.

Briefly, a semaphore is a construct that allows some threads to access a fixed set of resources in parallel. Always think of a semaphore as having a fixed number of permits to give out. Once all the permits are given out, requesting threads, need to wait for a permit to be returned before proceeding forward.

Your task is to implement a semaphore which takes in its constructor the maximum number of permits allowed and is also initialized with the same number of permits.

Solution

Given the above definition we can now start to think of what functions our Semaphore class will need to expose. We need a function to "gain the permit" and a function to "return the permit".

1. `acquire()` function to simulate gaining a permit

2. `release()` function to simulate releasing a permit

The constructor accepts an integer parameter defining the number of

permits available with the semaphore. Internally we need to store a count which keeps track of the permits given out so far.

The skeleton for our Semaphore class looks something like this so far.

```java
public class CountingSemaphore {

    int usedPermits = 0; // permits given out
    int maxCount;   // max permits to give out

    public CountingSemaphore(int count) {
        this.maxCount = count;
    }

    public synchronized void acquire() throws InterruptedException {
    }

    public synchronized void release() throws InterruptedException {
    }
}
```

Note we have added the `synchronized` keyword to both the class methods. Adding the synchronized keyword causes only a single thread to execute either of the methods. If a thread is currently executing `acquire()` then another thread can't execute `release()` on the same semaphore object.

Note this will guarantee that the `usedPermits` variable is correctly incremented or decremented.

The astute observer would question why we don't take the locking to finer grained level and use java's lock so that multiple threads can call either of the two functions. With `synchronized` only one thread can call either release or acquire. However, the counter to that is, even with finer grained locking the entire code blocks within the two methods will be guarded by a single lock and that would pretty much be the same as putting synchronized on the methods definitions.

Now let us fill in the implementation for our acquire method. When can a thread not be allowed to acquire a semaphore? When all the permits are out! This implies we'll need to `wait()` when `usedPermits == maxCount` If this condition isn't true we simply increment `usedPermits` to simulate giving out a permit.

The implementation of the acquire method appears below. Note that we are also `notify()`-ing at the end of the method. We'll talk shortly, about why we need it.

```
1.  public class CountingSemaphore {
2.
3.      int usedPermits = 0; // permits given out
4.      int maxCount; // max permits to give out
5.
6.      public CountingSemaphore(int count) {
7.          this.maxCount = count;
8.      }
9.
10.     public synchronized void acquire() throws InterruptedExceptio
n {
11.
12.         while (usedPermits == maxCount)
13.             wait();
14.
15.         usedPermits++;
16.         notify();
17.     }
18.
19.     public synchronized void release() throws InterruptedExceptio
n {
20.     }
21. }
```

Implementing the release method should require a simple decrement of the `usedPermits` variable. However when should we block a thread from proceeding forward like we did in `acquire()` method? If `usedPermits == 0` then it won't make sense to decrement `usedPermits` and we should block at this condition.

This might seem counter-intuitive, you might ask why would someone call `release()` before calling `acquire()` - This is entirely possible since semaphore can also be used for signalling between threads. A thread can call `release()` on a semaphore object before another thread calls `acquire()` on the same semaphore object. There is no concept of ownership for a semaphore ! Hence different threads can call acquire or

release methods as they deem fit.

This also means that whenever we decrement or increment the `usedPermits` variable we need to call `notify()` so that any waiting thread in the other method is able to move forward. The full implementation appears below

```
1.  public class CountingSemaphore {
2.
3.      int usedPermits = 0; // permits given out
4.      int maxCount; // max permits to give out
5.
6.      public CountingSemaphore(int count) {
7.          this.maxCount = count;
8.
9.      }
10.
11.     public synchronized void acquire() throws InterruptedException {
12.
13.         while (usedPermits == maxCount)
14.             wait();
15.
16.         usedPermits++;
17.         notify();
18.     }
19.
20.     public synchronized void release() throws InterruptedException {
21.
22.         while (usedPermitsls == 0)
23.             wait();
24.
25.         usedPermits--;
26.         notify();
27.     }
28. }
```

As a follow-up, notice that we increment/decrement the `usedPermits` variable on **line 16** and **25** respectively and then call `notify()`. Does it matter if we switch the order of the two statements, i.e. call `notify()` first and then manipulate `usedPermits` ? The answer is no.

When `notify()` is called, the executing thread is still synchronized on the

semaphore object and any other thread will not be scheduled until the

executing thread exists the `release()` or `acquire()` method and by then the `usedPermits` variable has already been incremented or decremented even though that is the last statement in each method.

Also, remember that semaphores solve the problem of missed signals between cooperating threads. Imagine a producer/consumer application where the producer wants to notify the consumer of available content for consumption. The producer can call `acquire()` on a binary semaphore (one with max permits = 1) and the consumer can call `release()` before attempting to consume. This way the "signal" is in a sense stored for the consumer whenever the producer produces something.

Complete Code

The complete code appears below along with a test. Note how we acquire and release the semaphore in different threads in different methods, something not possible with a mutex. Thread t1 always acquires the semaphore while thread t2 always releases it. The semaphore has a max permit of 1 so you'll see the output interleaved between the two threads. You might see the print statements from the two threads not interleave each other and may appear twice in succession. This is possible because of how threads get scheduled for execution and also because we start with an unused permit.

The astute reader would also observe that the given solution will always block if the semaphore is initialized with zero permits

```
class Demonstration {
    public static void main( String args[] ) throws InterruptedException {

        final CountingSemaphore cs = new CountingSemaphore(1);

        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                try {
```

```java
                try {
                    for (int i = 0; i < 5; i++) {
                        cs.acquire();
                        System.out.println("Ping " + i);
                    }
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread t2 = new Thread(new Runnable() {

            @Override
            public void run() {
                for (int i = 0; i < 5; i++) {
                    try {
                        cs.release();
                        System.out.println("Pong " + i);
                    } catch (InterruptedException ie) {

                    }
                }
            }
        });

        t2.start();
        t1.start();
        t1.join();
        t2.join();
    }
}

class CountingSemaphore {

    int usedPermits = 0;
    int maxCount;

    public CountingSemaphore(int count) {
        this.maxCount = count;

    }

    public synchronized void acquire() throws InterruptedException {

        while (usedPermits == maxCount)
            wait();

        notify();
        usedPermits++;

    }

    public synchronized void release() throws InterruptedException {

        while (usedPermits == 0)
            wait();

        usedPermits--;
        notify();
    }
}
```