Unisex Bathroom Problem

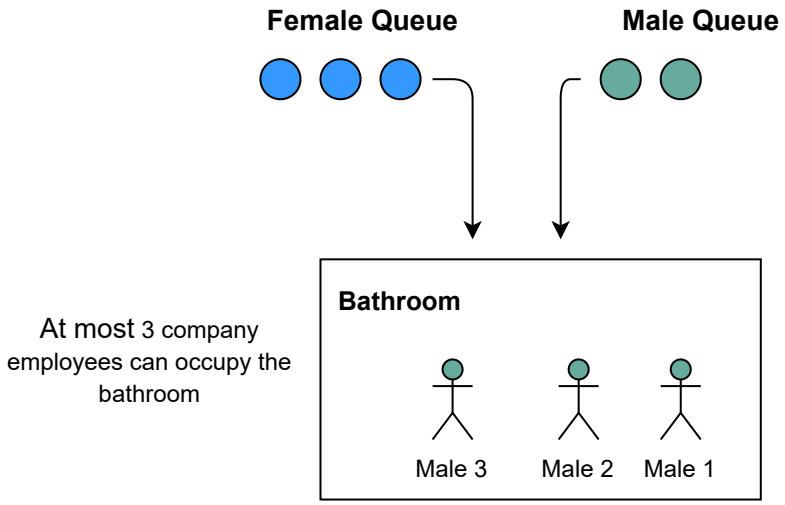
A synchronization practice problem requiring us to synchronize the usage of a single bathroom by both the genders.

Problem

A bathroom is being designed for the use of both males and females in an office but requires the following constraints to be maintained:

- There cannot be men and women in the bathroom at the same time.
- There should never be more than three employees in the bathroom simultaneously.

The solution should avoid deadlocks. For now, though, don't worry about starvation.



Unisex Bathroom Problem

Solution

First let us come up with the skeleton of our Unisex Bathroom class. We

want to model the problem programmatically first. We'll need two APIs,

one that is called by a male to use the bathroom and another one that is called by the woman to use the bathroom. Initially our class looks like the following

```
public class UnisexBathroom {

    void maleUseBathroom(String name) throws InterruptedException {
    }

    void femaleUseBathroom(String name) throws InterruptedException {
    }
}
```

Let us try to address the first problem of allowing either men or women to use the bathroom. We'll worry about the max employees later. We need to maintain state in a variable which would tell us which gender is currently using the bathroom. Let's call this variable `inUseBy`. To make the code more readable we'll make the type of the variable `inUseBy` a string which can take on the values men, women or none.

We'll also have a method `useBathroom()` that'll mock a person using the bathroom. The implementation of this method will simply sleep the thread using the bathroom for some time.

Assume there's no one in the bathroom and a male thread invokes the `maleUseBathroom()` method, the thread has to check first whether the bathroom is being used by a female thread. If it is indeed being used by a female, then the male thread has to wait for the bathroom to be empty. If the male thread already finds the bathroom empty, which in our scenario it does, the thread simply updates the `inUseBy` variable to **"MEN"** and proceeds to use the bathroom. After using the bathroom, however, it must let any waiting female threads know that it is done and they can now use the bathroom.

The astute reader would immediately realize that we'll need to guard the variable `inUseBy` since it can possibly be both read and written to by

variable `inUseBy` since it can possibly be both read and written to by different threads at the same time. Does that mean we should mark our methods as `synchronized`? The wary reader would know that doing so would essentially make the threads serially access the methods, i.e., if one male thread is accessing the bathroom, then another one can't access the bathroom even though the problem says that more than one male should be able to use the bathroom. This requires us to take synchronization to a finer granular level rather than implementing it at the method level. So far what we discussed looks like the below when translated into code:

```
0. public class UnisexBathroom {
1.
2.    static String WOMEN = "women";
3.     static String MEN = "men";
4.     static String NONE = "none";
5.
6.     String inUseBy = NONE;
7.     int empsInBathroom = 0;
8.
9.     void useBathroom(String name) throws InterruptedException {
10.        System.out.println(name + " using bathroom. Current employ
ees in bathroom = " + empsInBathroom);
11.        Thread.sleep(10000);
12.        System.out.println(name + " done using bathroom");
13.    }
14.
15.    void maleUseBathroom(String name) throws InterruptedException
{
16.
17.        synchronized (this) {
18.            while (inUseBy.equals(WOMEN)) {
19.                // The wait call will give up the monitor associat
ed
20.                // with the object, giving other threads a chance
to
21.                // acquire it.
22.                this.wait();
23.            }
24.            empsInBathroom++;
25.            inUseBy = MEN;
26.        }
27.
28.        useBathroom(name);
29.
```

```
30.         synchronized (this) {
31.             empsInBathroom--;
32.
33.             if (empsInBathroom == 0) inUseBy = NONE;
34.             // Since we might have just updateded the value of
35.             // inUseBy, we should notifyAll waiting threads
36.             this.notifyAll();
37.         }
38.     }
39.
40.     void femaleUseBathroom(String name) throws InterruptedExceptio
n {
41.
42.         synchronized (this) {
43.             while (inUseBy.equals(MEN)) {
44.                 this.wait();
45.             }
46.             empsInBathroom++;
47.             inUseBy = WOMEN;
48.         }
49.
50.         useBathroom(name);
51.
52.         synchronized (this) {
53.             empsInBathroom--;
54.
55.             if (empsInBathroom == 0) inUseBy = NONE;
56.             // Since we might have just updateded the value of
57.             // inUseBy, we should notifyAll waiting threads
58.             this.notifyAll();
59.         }
60.     }
61.}
```

The code so far allows any number of men or women to gather in the bathroom. However, it allows only one gender to do so. The methods are mirror images of each other with only gender-specific variable changes. Let's discuss the important portions of the code.

- **Lines 17-26:** Since java monitors are mesa monitors, we use a while

loop to check for the variable `inUseBy`. If it is set to **MEN** or **NONE**

then, we know the bathroom is either empty or already has men and therefore it is safe to proceed ahead. If the `inUseBy` is set to **WOMEN**, then the male thread, invokes `wait()` on **line 23**. Note, the thread would ***give up the monitor for the object on which it is synchronized*** thus allowing other threads to synchronize on the same object and maybe update the `inUseBy` variable

- **Line 28** has no synchronization around it. If a male thread reaches here, we are guaranteed that either the bathroom was already in use by men or no one was using it.
- **Lines 30-37:** After using the bathroom, the male thread is about to leave the method so it should remember to decrement the number of occupants in the bathroom. As soon as it does that, it has to check if it were the last member of its gender to leave the bathroom and if so then it should also update the `inUseBy` variable to **NONE**. Finally, the thread notifies any other waiting threads that they are free to check the value of `inUseBy` in case it has updated it. **Question**: Why did we use `notifyAll()` instead of `notify()`?

Now we need to incorporate the logic of limiting the number of employees of a given gender that can be in the bathroom at the same time. ***Limiting access, intuitively leads one to use a semaphore.*** A semaphore would do just that - limit access to a fixed number of threads, which in our case is 3.

Complete Code

The complete code appears below:

```
import java.util.concurrent.Semaphore;

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

```
        UnisexBathroom.runTest();
    }
}


class UnisexBathroom {

    static String WOMEN = "women";
    static String MEN = "men";
    static String NONE = "none";

    String inUseBy = NONE;
    int empsInBathroom = 0;
    Semaphore maxEmps = new Semaphore(3);

    void useBathroom(String name) throws InterruptedException {
        System.out.println("\n" + name + " using bathroom. Current employees in bathroom = "
        Thread.sleep(3000);
        System.out.println("\n" + name + " done using bathroom " + System.currentTimeMillis()
    }

    void maleUseBathroom(String name) throws InterruptedException {

        synchronized (this) {
            while (inUseBy.equals(WOMEN)) {
                this.wait();
            }
            maxEmps.acquire();
            empsInBathroom++;
            inUseBy = MEN;
        }

        useBathroom(name);
        maxEmps.release();

        synchronized (this) {
            empsInBathroom--;

            if (empsInBathroom == 0) inUseBy = NONE;
            this.notifyAll();
        }
    }

    void femaleUseBathroom(String name) throws InterruptedException {

        synchronized (this) {
            while (inUseBy.equals(MEN)) {
                this.wait();
            }
            maxEmps.acquire();
            empsInBathroom++;
            inUseBy = WOMEN;
        }

        useBathroom(name);
        maxEmps.release();

        synchronized (this) {
            empsInBathroom--;

            if (empsInBathroom == 0) inUseBy = NONE;
            this.notifyAll();
        }
```

```
    }

    public static void runTest() throws InterruptedException {

        final UnisexBathroom unisexBathroom = new UnisexBathroom();

        Thread female1 = new Thread(new Runnable() {
            public void run() {
                try {
                    unisexBathroom.femaleUseBathroom("Lisa");
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread male1 = new Thread(new Runnable() {
            public void run() {
                try {
                    unisexBathroom.maleUseBathroom("John");
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread male2 = new Thread(new Runnable() {
            public void run() {
                try {
                    unisexBathroom.maleUseBathroom("Bob");
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread male3 = new Thread(new Runnable() {
            public void run() {
                try {
                    unisexBathroom.maleUseBathroom("Anil");
                } catch (InterruptedException ie) {

                }
            }
        });

        Thread male4 = new Thread(new Runnable() {
            public void run() {
                try {
                    unisexBathroom.maleUseBathroom("Wentao");
                } catch (InterruptedException ie) {

                }
            }
        });

        female1.start();
        male1.start();
        male2.start();
        male3.start();
        male4.start();
```

```
        female1.join();
        male1.join();
        male2.join();
        male3.join();
        male4.join();

    }
}
```

If you look at the program output, you'd notice that the current employees in the bathroom at one point is printed out to be 4 when the max allowed is 3. This is just an outcome of how the code is structured, read on below for an explanation.

In our test case we have four males and one female aspiring to use the bathroom. We let the female thread use the bathroom first and then let all the male threads loose. From the output, you'll observe, that no male thread is inside the bathroom until Lisa is done using the bathroom. After that, three male threads get access to the bathroom at the same instant. The fourth male thread is held behind, till one of the male thread exits the bathroom.

We acquire the semaphore from within the synchronized block and in case the thread blocks on acquire ***it doesn't give up the monitor, for the object*** which implies that `inUseBy` and `empsInBathroom` won't be modified till this blocked thread gets out of the synchronized block. This is a very subtle and important point to understand.

Imagine, if there are already three men in the bathroom and a fourth one comes along then he gets blocked on **line#31**. This thread still holds the bathroom object's monitor, when it becomes dormant due to non-availability of permits. This prevents any female thread from changing the `inUseBy` to **WOMEN** under any circumstance nor can the value of `empsInBathroom` be changed.

Next note the threads returning from the `useBathroom` method, release the semaphore. We must release the semaphore here because if we do not then the blocked fourth male thread would never release the object's

monitor and the returning threads will never be able to access the second synchronization block.

On releasing the semaphore, the blocked male thread will increment the `empsInBathroom` variable to 4, before the thread that signaled the semaphore enters the second synchronized block and decrements itself from the count. It is also possible that male threads pile up before the second synchronized block, while new arriving threads are chosen by the system to run through the first synchronized block. In such a scenario, the count `empsIBathroom` will keep increasing as threads returning from the bathroom wait to synchronize on the `this` object and decrement the count in the second synchronization block. Though eventually, these threads will be able to synchronize and the count will reach zero.

As an alternative, we can put the statement `maxEmps.acquire()` on **line # 35** instead of **line # 31** and the program will continue to work correctly. However, then the variable `empsInBathroom` can potentially take up a value equal to the number of waiting threads. In our current version, the max value the variable `empsInBathroom` can take up is 4.

To prove the correctness of the program, you'll need to convince yourself that the variables involved are all being atomically manipulated.

Also, note that this solution isn't fair to the genders. If the first thread to get bathroom access is male, and before it's done using the bathroom, a steady stream of male threads start to arrive for bathroom use, then any waiting female threads will starve.

Follow up

- Try writing a solution in which there's no possibility of starvation for threads of either gender.