Unisex Bathroom Problem

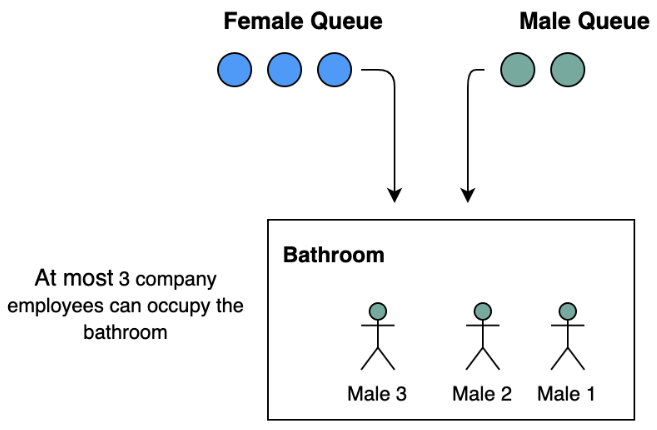
In this lesson, we will look at a synchronization practice problem requiring us to synchronize the usage of a single bathroom by both genders.

Unisex Bathroom 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.



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 that is called by a female to use the bathroom. Initially, our class looks like the following:

```
class UnisexBathroomProblem

  def initialize()
  end

  def useBathroom(name)
  end

  def maleUseBathroom(name)

  end

  def femaleUseBathroom(name)
  end

end
```

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 to track which gender is currently using the bathroom. Let's call this variable `inUseBy`. The variable `inUseBy` will take on the values **female, male or none** to denote the gender currently using the bathroom.

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 **"male"** 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 different threads at the same time. We can use a `Mutex` variable for it. We'll also use a condition variable to allow us to `wait()` threads of one gender while threads of the other gender are using the bathroom. We'll also introduce a variable `empsInBathroom` to keep count of employees currently in the bathroom. Note this variable isn't required to solve the problem but useful to test our solution. Using the above discussion we can craft the `maleUseBathroom()` method and the rest of the code as follows:

```
class UnisexBathroomProblem

  def initialize()
    @inUseBy = "none"
    @empsInBathroom = 0
    @cond = ConditionVariable.new
    @mutex = Mutex.new
  end

  def useBathroom(name)
    # simulate using a bathroom
    puts "\n#{name} is using the bathroom. #{@empsInBathroom} employees in bathroom"
    sleep(1)
    puts "#{name} is done using the bathroom"
  end

  def maleUseBathroom(name)

    @mutex.lock()
    while @inUseBy == "female"
      @cond.wait(@mutex)
    end
    @maxEmpsSem.acquire()
    @empsInBathroom += 1
    @inUseBy = "male"
    @mutex.unlock()

    useBathroom(name)
    @mutex.lock()
    @empsInBathroom -= 1
    if @empsInBathroom == 0
      @inUseBy = "none"
    end
    @cond.broadcast()
    @mutex.unlock()
  end
```

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 20-27:** After acquiring the condition variable on **line#21** we use a while loop to check for the predicate `inUseBy`. If it is set to **male** 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 **female**, then the male thread, invokes `wait()` on **line#21**. Note, the thread would ***give up the lock associated with the condition variable*** thus allowing other threads to acquire the condition variable and possibly 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 31-37:** 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 `broadcast()` instead of `signal()`?

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.

main.rb

CountingSemaphore.rb

```ruby
class CountingSemaphore

  def initialize(maxPermits)
    @maxPermits = maxPermits
    @givenOut = 0
    @monitor = Monitor.new
    @condVar = @monitor.new_cond()
  end

  def acquire()
    @monitor.enter()

    while @givenOut == @maxPermits
      @condVar.wait()
    end

    @givenOut += 1
    @condVar.broadcast()

    @monitor.exit()
  end

  def release()
    @monitor.enter()

    while @givenOut == 0
      @condVar.wait()
    end

    @givenOut -= 1
    @condVar.broadcast()

    @monitor.exit()
  end
end
```

If you look at the program output, you'd notice that the number of current employees in the bathroom is printed out to be greater than 3 at times even though the maximum allowed employees in the bathroom are 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 until one of the male thread exits the bathroom.

We acquire the semaphore from within the synchronized block and in case a thread blocks on `acquire()`, ***it doesn't give up the mutex lock we associate with the condition variable*** which implies that variables `inUseBy` and `empsInBathroom` don't get modified until this blocked thread gets out of the synchronized block. This is a very subtle and important point to understand.

Imagine that there are already three men in the bathroom and a fourth one comes along then he gets blocked on **line 26**. This fourth thread still holds the condition variable's lock, when it becomes dormant due to the non-availability of permits. This prevents any female thread from changing the `inUseBy` to **female** under any circumstances nor can the value of `empsInBathroom` be changed.

Next, note the threads returning from the `use_bathroom()` 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 mutex object and the returning threads from the bathroom will never be able to access the second synchronization block.

On releasing the semaphore, the blocked fourth male thread will increment the `empsInBathroom` variable to 4, before the thread that signaled the semaphore enters the second synchronized block and decrements `empsInBathroom` back to 3. However, it is also possible that male threads pile up before the second synchronized block, while new arriving threads are chosen by the system to acquire the mutex variable first and run through the first synchronization block. In such a scenario, the count `empsInBathroom` would keep increasing as threads returning from the bathroom wait to acquire the mutex variable and decrement the count in the second synchronization block. Though eventually, these threads will acquire the mutex variable and the count would reach zero.

To prove the correctness of the program, you'll need to convince yourself that the variables involved are 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 keep arriving to use the bathroom, then any waiting female threads will starve.