

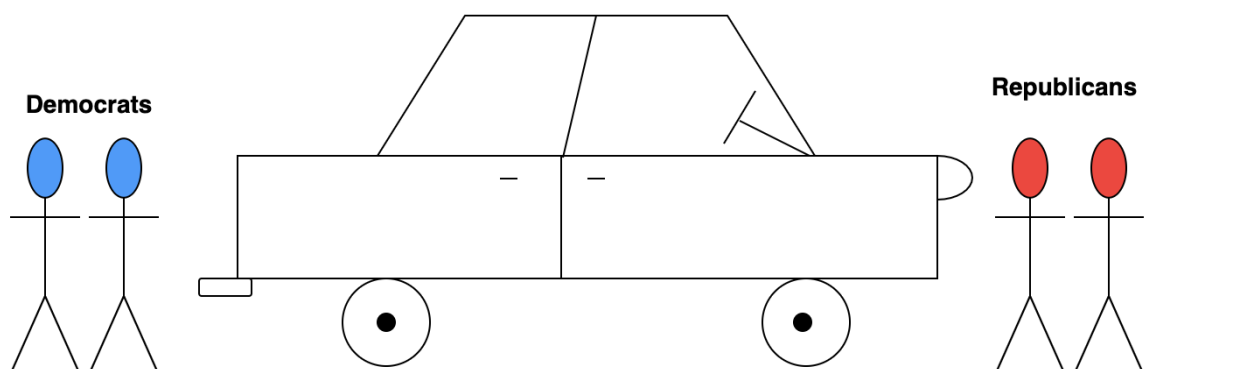
# Uber Ride Problem

This lesson solves the constraints of an imaginary Uber ride problem where Republicans and Democrats can't be seated as a minority in a four passenger car.

## Uber Ride Problem

Imagine at the end of a political conference, republicans and democrats are trying to leave the venue and ordering Uber rides at the same time. However, to make sure no fight breaks out in an Uber ride, the software developers at Uber come up with an algorithm whereby either an Uber ride can have all democrats or republicans or two Democrats and two Republicans. All other combinations can result in a fist-fight.

Your task as the Uber developer is to model the ride requestors as threads. Once an acceptable combination of riders is possible, threads are allowed to proceed to ride. Each thread invokes the method `seated()` when selected by the system for the next ride. When all the threads are seated, any one of the four threads can invoke the method `drive()` to inform the driver to start the ride.



First let us model the problem as a class. We'll have two methods one called by a Democrat and one by a Republican to get a ride home. When either one gets a seat on the next ride it'll call the `seated()` method.

To make up an allowed combination of riders, we'll need to keep a count of Democrats and Republicans who have requested rides. We create two variables for this purpose and modify them within a lock/mutex. In this problem, we'll use an object of the `Mutex` class when manipulating counts for democrats and republicans.

Realize that we'll also need a barrier where all the four threads, that have been selected for the Uber ride arrive at, before riding away in the same car. This is analogous to the four riders being seated in the car and the doors being shut.

Once the doors are shut, one of the riders has to tell the driver to drive which we simulate with a call to the `drive()` method. Note exactly one thread makes the shout-out to the driver to `drive()`.

The initial class skeleton looks like the following:

```
class UberSeatingProblem

  def initialize()
    @democratsCount = 0
    @democratsWaiting = CountingSemaphore.new(4, 4)
    @republicansCount = 0
    @republicansWaiting = CountingSemaphore.new(4, 4)
    @lock = Mutex.new
    @barrier = Barrier.new(4)
    @rideCount = 0
  end

  def drive()
    @rideCount += 1
    puts "\nUber ride #{@rideCount} filled and on its way"
  end

  def seated(party)
    puts "\n #{party} #{Thread.current.__id__} seated"
  end
end
```

```
def seatDemocrat()  
  
end  
  
def seatRepublican()  
  
end  
end
```

Let's focus on the `seatDemocrat()` method first. For simplicity imagine the first thread is a democrat and invokes `seatDemocrat()`. Since there's no other rider available, it should be put to wait. We can use a semaphore to make this thread wait. We'll not use a barrier, because we don't know what party loyalty the threads arriving in the future would have. It might be that the next four threads are all Republican and this Democrat isn't placed on the next Uber ride. To differentiate between waiting Democrats and waiting Republicans, we'll use two different semaphores `democratsWaiting` and `republicansWaiting`. Our first democrat thread will `lock()` the `lock` mutex variable, find that no other riders exist, release the `lock` object and go on to wait at the `democratsWaiting` semaphore. Note that we initialize both the semaphores with max permits four and with zero permits available.

Now it's easy to reason about how we select the threads for a ride. A democrat thread has to check the following cases:

- If there are already 3 waiting democrats, then we signal the `democratsWaiting` semaphore three times so that all these four democrats can ride together in the next Uber ride.
- If there are two or more republican threads waiting and at least two democrat threads (including the current thread) waiting, then the current democrat thread can signal the `republicansWaiting` semaphore twice to release the two waiting republican threads and signal the `democratsWaiting` semaphore once to release one more democrat thread. Together the four of them would make up the next ride consisting of two Republicans and two Democrats.
- If the above two conditions aren't true then the current democrat thread should simply wait itself at the `democratsWaiting` semaphore

and release the mutex so that other threads can now enter the critical sections.

The logic we discussed so far is translated into code below:

```
def seatDemocrat()
  rideLeader = false

  @lock.lock()

  @democratsCount += 1

  if @democratsCount == 4
    # release 3 democrats to ride along
    @democratsWaiting.release()
    @democratsWaiting.release()
    @democratsWaiting.release()
    rideLeader = true
    @democratsCount -= 4

  elsif @democratsCount == 2 and @republicansCount >= 2
    # release 1 democrat and 2 republicans
    @democratsWaiting.release()
    @republicansWaiting.release()
    @republicansWaiting.release()
    rideLeader = true

    # remember to decrement the count of dems and repubs
    # selected for next ride
    @democratsCount -= 2
    @republicansCount -= 2

  else
    # can't form a valid combination, keep waiting and release lock
    @lock.unlock()
    @democratsWaiting.acquire()
  end

  seated("Democrat")
  @barrier.await()

  if rideLeader == true
    drive()
    @lock.unlock()
  end
end
```

The thread that signals other threads to come along for the ride marks itself as the **rideLeader**. This thread is responsible for informing the driver to **drive()**. We can come up with some other criteria to choose the rider leader but given the logic we implemented, it is easiest to make the thread that determines an acceptable ride combination as the ride leader.

Note the use of barrier on **line#35**. All threads that form the next ride congregate at this point before moving forward.

The republicans' `seatRepublican()` method is analogous to the `seatDemocrat()` method.

main.rb

CountingSemaphore.rb

Barrier.rb

```
class CountingSemaphore

  def initialize(maxPermits, initialAvailablePermits)
    @maxPermits = maxPermits
    @givenOut = maxPermits - initialAvailablePermits
    @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
```



The output of the above program shows we create ten of each democratic and republican rider threads. The output must contain at least one ride

evenly split between the two types of riders. You can change the number of democrats and republicans on **lines#113-114**. However, the program run may report an error if all republican and democrat riders can't be found valid combinations for a ride.

The astute reader may wonder what factor determines that a ride is evenly split between members of the two parties or entirely made up of the members of the same party, given enough riders exist that both combinations can be possible. The key is to realize that each thread enters the critical sections `seatDemocrat()` or `seatRepublican()` one at a time because of the lock at the beginning of the two methods. Whether a ride is evenly split between the two types of riders or consists entirely of one type of riders depends upon the order in which the threads enter the critical section. For instance, if we create four democrat and four republican threads then we can either get two rides each split between the two political parties or two rides each made of the same political party. If the first four threads to sequentially enter the critical sections are democrat, then the two rides will be made up of either entirely democrats or republicans. Since threads are scheduled for execution non-deterministically, we can't be certain what would be the makeup of each ride.