Concurrent Programming - assignment 3

Matej Majtan - s184457, Jun Chen - s202781

November 2020

1 Problem 1

The problem 1 concerns creation of a monitor based gate for the car problem instead of the given semaphore based solution. The monitor based solution uses the Java built-in `synchronized` method modifier. This allows all the methods of the `MonitorGate` to be mutually exclusive and avoid mishandling the shared variable `isopen`. This variable is used to keep track of the state of the gate. When the gate is closed this variable is set to `false`. When the car tries to pass the gate this variable controls whether the car can pass or in the case that the gate is closed it makes the car wait until it is released. The release of the car happens when the gate gets open as the variable is set to `true` and the gate notifies the thread waiting to pass.

2 Problem 2

The given solution for `NaiveBarrier` has several flaws that can cause issues when executing. First of them concerns the issue regarding releasing of the barrier, when the cars can get stuck and not continue after all arriving to the barrier. The cause of this issue is that the increment of the variable `arrived` is not protected. As this action is not an atomic operation and takes time to first read, then modify and finally write the variable, the program can possibly allow two cars to read from the same variable right after each other, before the first car manages to write the new value into the variable and causes the variable not to be updated properly. This means that even if all cars are at the barrier, they will not be released as the condition will never be reached.

The second issue can happen when cars can end up stuck at the barrier even though the barrier has been turned off. The reason is that between checking whether the barrier is `active` and executing the actual logic for the barrier, the thread needs to update the `arrived` variable and if in that time user decides to turn the barrier off, the car will still start to wait at the barrier. This can be easily tested by adding `Thread.sleep(100)` before or after the `arrived++` statement. This causes the program to take longer time to reach the barrier logic which helps with the manual test.

The test can be conducted by turning a barrier on and letting a car approach it. In the instance when the car gets to the barrier, which is approximately when the thread is sleeping, turn off the barrier and when the sleep timer is over the car is waiting at the barrier even without a barrier there.

Further there is another not protected race condition. This race condition is the `arrived` counter, which is being access from two places. One of the places is the `sync` method where the counter is updated and checked in the *if* statement and inside the `off` method where it is reset. This race condition can cause inconsistencies when a car is checking the *if* statement and in the same instance the user turns off the barrier.

This solution is not robust towards spurious wakeups. The reason is that whenever a spurious wake up happens the program is not checking whether the condition really holds or not. The exact place in the code where this happens is when the `arrived` variable is being tested for being at the threshold value (by default 9), then the car waits or notifies other threads depending on the outcome. In the case that a waiting car gets notified it simply continues in execution instead of checking whether the condition is actually *true*. This can be tested by creating a `threshold` variable used in the condition and using the `set` method to set it. When the threshold is set to a larger number and smaller amount of cars is at the barrier, then if the user updates the threshold value, nothing will happen. Therefore it is necessary to inform the cars about being able to move, which can be done by notifying all waiting cars when the threshold value is changed. However this brings up the issue that the number of arrived cars is not always smaller than the threshold and the condition is not checked before continuing with execution.

Fixing this solution without taking the spurious wakeups into consideration can be done by simply expanding the `synchronized` block onto the whole scope of the methods. This way any variable access as well as decision making is protected. Same outcome can be achieved by using `synchronized` modifier for all the methods. The first solution can be seen in the attached code.

3 Problem 3

As mentioned in problem 2 that the spurious wakeups may occur when the threshold is changed, it can happen that the barrier would not release cars according to the newly updated threshold condition. In order to solve this problem, method `set()` has to signal awaiting cars which are waiting for condition `isOverThreshold` in method `sync`. A while loop is used to check whether the number of current arrived cars is higher than the latest threshold, if that is not the case then the car would wait for signal of condition `isOverThreshold`. The while loop ensures that when a spurious wakeup happens, the condition will be rechecked before moving on.

Another issue of this problem is that only the last left car should reset number of arrived cars

to 0 otherwise the remaining arrived cars cannot leave, as the condition that is being rechecked would no longer hold. Therefore once condition `isOverThreshold` is satisfied, the thread counts the number of ready cars out of the `isOverThreshold` condition. The cars then await for the *isLastCarArrived* to be released. As soon as the last car gets moved out of the `isOverThreshold` condition, then the signal for the *isLastCarArrived* condition is released and the number of arrived cars is reset.

4 Problem 5

In order to remove particular cars, firstly the corresponding conductors need to be interrupted otherwise they might be stuck at waiting within thread which can cause the whole program to freeze. Secondly the selected cars need to leave the current position and the position they are about to go to by using function `leave` in class `Field` in order to allow other cars to enter those fields. On top of that if a car is in the *alley* and it is to be removed, it has to leave the *alley* to avoid blocking cars from the opposite direction entering the *alley*. This is achieved by creating a boolean variable `inAlley` and a semaphore `inAlleySem` in class `Conductor`. Boolean variable `inAlley` updates when a car enters and leaves the *alley*, and it is protected by semaphore `inAlleySem`. Therefore function `removeCar` can check whether the removed car is in the *alley* and use function `leave` under class `Alley` if that is the case. Furthermore the car has to `deregister` from the *car display* object which is referenced to GUI. Lastly the corresponding conductors of removed cars are set to `null`.

For restoring removed cars, function `restoreCar` ensures the conductor with the same number as the target car is `null`. This check prevents duplicates. Then a new `Conductor` object will be created and started. As the same way to remove the car from GUI, restoring a car on GUI requires the car being registered from the *car display* object.

5 Conclusion

The focus of the assignment was to demonstrate our knowledge of monitors. The solution for monitor based `Gate` implementation turned out to be much simpler than using semaphores while keeping the integrity of the program. Later a problem of `NaiveBarrier` was discussed where the problems with the implementation were pointed out and a simple solution was suggested. However this solution didn't cover the problem of spurious wakeups, which has been solved by the use of two instances of the `Condition` class and by using two layer synchronization. As last discussed point was the problem of removing and restoring cars in the application. This was achieved by slightly modifying the `Conductor` class and implementing the respective functions. Overall all necessary requirements were met in all the problems.