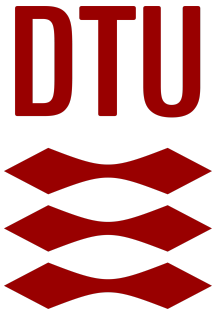

Mandatory Assignment 3

(02158) Cuncurrent Programming

Anton Lage
s191270

Daniel Schütt
s194592

November 8, 2022

Problem 1

In this problem we had to translate the semGate.java, which uses semaphores to control access through each cars gate, into monGate.java, which uses a monitor to achieve the same functionality. This was done with a simple boolean value indicationg weather or not the gate is open, functions to change this value, and a final function for passing through the gate. If the gate is closed cars are put in the condition queue (only one car is using each gate, so there will never be more than one car in the queue). When the gate is opened the queue is notified, and if there are a waiting car it will be released. If the gate is already open when the car arrives, it will just pass through without waiting.

Problem 2

Problem 2.1

The reason as to why the solution might fail to release the barrier when all cars have arrived is due to the `sync` function. Not the entire function is synchronized, despite being a critical region. This means more than one car can call the `sync` function, which then unsynchronizedly increments the `arrived` variable. Since incrementing a variable is not an atomic action, and since this is a critical region, the program will be error prone.

This means the `arrived` variable might no longer represent the actual amount of arrived cars. If this is the case the barrier will never go down, since no more arrive will never be properly incremented to the set threshold, leaving all cars waiting forever.

Problem 2.2

If a process, P1, checks if the barrier is active, but before it continues to incrementing `arrive` and the synchronized part of the method, a process, P2, turning the barrier off takes over execution. Then P2 will release all waiting cars. But at that point in time P1 will not yet have made the call to wait, and is therefore not receiving the signal that the barrier has been turned off (i.e to continue), but it has checked to see if the barrier was turned on, and is therefore going to wait once it resumes execution.

If a `Thread.Sleep(2000)` statement is inserted between `if (!active) return;` and `arrived++;` statements, and the barrier is turned off within two seconds of a car arriving at the barrier, the car will not continue after the barrier is turned off.

Problem 2.3

Another way the program can malfunction due to a race condition is with the `off` function. If the `off` function is called at about the same time as a car enters the `sync` function, then the `off` function can potentially set `arrived` to 0. Then the car, which has already loaded the `arrived` variable, will set `arrived` to another number which the car has incremented.

The next time the barrier is activated, the `arrived` variable will still be set to said number. This means that there are fewer children who will be waiting before the barrier is released next time.

Problem 2.4

This solution is not robust towards spurious wakeups, which is due to the `sync` function. When a car is stopped by a barrier, the only thing stopping it from driving is the fact that it is waiting. If a spurious wakeup were to happen, then the car would stop waiting, and simply go over the barrier, since it does not recheck its waiting condition.

We can simulate the spurious wakeups by adding a `threshold` variable. When a car waits for the barrier, and the amount of waiting cars equal the threshold set by the user, then before waiting the car uses the `notify` function. This means the car wakes up a random waiting car, simulating a spurious wakeup.

Problem 2.5

Most of the issues in the file NaiveBarrier stems from the functions only being partially synchronized. If the functions `set, on, off` and `sync` were synchronized entirely then every problem described here except spurious wakeup would be fixed.

the issues coming from spurious wakeup wouldn't be fixed, as that problem stems from how the cars are waiting, and not synchronization errors which are handled in this problem.

Problem 3

To safeguard against spurious wakeups, we put the `wait` function in a loop, so that every time a process wakes up, it rechecks its wait condition. Additionally, we also implemented a new wait which is now a boolean flag instead of a counter. Each process has its own boolean flag, which are all set to true by the last arriving process, before that process calls `notifyAll`. The boolean flags are then cleared by its corresponding process, when it leaves the monitor, so that it is not able to cross the barrier again, until after all threads have made it back to the barrier, and the flags have been reset.

Problem 5

When removing the car, it isn't enough to merely deregister the car. Calling the `deregister` function does remove the car from the track, however it does not release the resources that car possesses in the moment of removal, which means the missing car will continue to block the other cars.

Therefore, before removal of the car, the allocated locks have to be released (calling `field.leave` on all fields the car holds the lock to), and if the car is in the alley, this has to be accounted for in the entrance protocol (calling `alley.leave()`)

This turned out to be not so trivial, but in the end we believe we made it work.

First of all, the enter and leave functionality in the `run()` method of the conductor was moved into the following dedicated synchronized methods: `enterFirstField()`, `enterNextField()`, `enterAlley()` and `exitFieldProcedure()`. It these methods, aside from calling the actual enter and leave methods, state variables are also updated. The methods are accompanied by similar synchronized methods for reading the state variables, which are as follows: `isAlleyLocked()`, `isCurrentFieldLocked()` and `isNextFieldLocked()`.

This enables us to do two things. The first one is the make sure the state is correct, when removing a car. When doing so, we can check if a car holds a given resource one at a time, without worrying about if the conductor is in the middle of updating the state, and if it does, release that resource.

The other thing it enables us to do, is interrupting the conductor thread if it is waiting for a resource, while still making sure the state is correct. For example: If the car is waiting to enter the alley, an has been put into a waiting queue, calling interrupt on the conductor thread will pull it out of waiting, and make it return through catch statements. On the other hand, if the conductor have just entered the alley, but have not yet updated the `inAlley` variable, calling interrupt will not do anything, and it will still finish updating the state correctly.

After releasing all resources, we remove the car from the UI via the `deregister` function, and set `conductor[no] = null`. This allows us to make a check if removing the same car again, so that we do not try and remove it again. Similarly, when restoring the car, we can check that no car exists, and just create a brand new one, like they are created the first time on program startup.

Conclusion

We have in this assignment worked with monitors in Java. While translating from semGate.java to monGate.java we got an impression of how much simpler a monitor can be, rather than implementing the same thing using semaphores.

We were then tasked with analyzing and remedying a solution to a barrier implementation, which was supposed to ensure that each car drove the same amount of rounds. The provided solution had a variety of problems, as shown in our paper under section 2, including a couple of race-conditions as well as it being vulnerable to spurious wake-ups. We remedied the solution against the race-conditions, but not against spurious wake-ups.

In problem 3, we where tasked with implementing a barrier solution, which was robust against spurious wake-ups. We implemented several changes compared to the solution in problem 2, and in the end we believe we have succeeded.

Problem 5 required us to implement removal and restoring of cars from the playground. This was, in our belief, to make us understand the importance and difficulty of releasing resources properly when done with them.