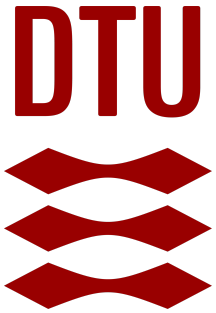

Mandatory Assignment 2

(02158) Cuncurrent Programming

Anton Lage
s191270

Daniel Schütt
s194592

October 11, 2022

Problem 1

In Problem 1, we had to implement an exit protocol for the critical region, as well as implementing the code for the Coordinator, controlling which process gets access to its critical section. This was to be done with two arrays of flags. One array called `enter` and one called `ok`. In the given entry protocol, a process sets its own enter flag (`enter[_pid] = 1`), and then proceeds to wait for an ok flag to be set, allowing it into the critical section (`await ok[_pid] == 1`).

The solution we came to, was to have the Coordinator look at each entry flag one at a time in a loop, and if it encounters one which is set, clear it and set the corresponding ok flag. It then waits for the process to clear its own ok flag, before moving on to checking the other processes entry flag. The exit protocol is therefore implemented by simply having the process in the critical section clear its own ok flag when the process finishes, allowing the Coordinator to proceed to look at other enter flags.

Our solution is shown in Lab2c.pml, and it passes both mutual exclusion safety check (i.e there is no assertion error), as well as pass the fairness test

```
ltl fair  []( (P[0]@entry) -> <> (P[0]@crit) )
```

stating that if process 0 is ever at its entry protocol, it will eventually be at its critical section.

Problem 2

In the conductor class' run method, before the car is driven to a field, the `field.enter()` method is called. Also, the `field.leave()` method is not called until after a car has been driven to a new field. This allows for treating each field as a critical region, and *being* in a field as a critical section, with it's own individual semaphore.

To solve the problem, an array of 11x12 semaphores is created and initialized to 1. This limits each field to containing at most one car, at any one time. When entering a new field the conductor thread calls `field.enter()`, in which the `P()` method on the semaphore, related to the field the car wants to enter, is called. This blocks the car from entering, until it can obtain the semaphore lock for the desired field. Likewise, when a car is leaving a field, it calls the `field.leave()` method, from which, the `V()` method of that field's semaphore is called, releasing the field for a potential other car to enter.

This does however introduce deadlock, since there is no regulation for going into the alley, and cars can freely enter from both sides whenever they arrive. But when cars enter from both sides, and they are not allowed to be in the same field (i.e drive into each other), the cannot pass by each other, and they are stuck.

Problem 3

When using `MonoAlley`, the alley is created as one critical region, where only one car is allowed at a time. This is done with a single semaphore, which the cars have to grab, in order to enter the alley. Because of this, cars have to wait until a car in the alley have completely exited the alley, before they can try to grab the semaphore.

This is not optimal (or as described: fun for the kids), since multiple cars going in the same direction could share the critical region at the same time, and thereby maximize efficiency. For a solution to this,

we move on to the use of `MultiAlley`

When using `MultiAlley` two semaphores are created for the alley; One for going one way (up) and one for going the other way (down). When there are no cars in the alley, and a car is about to go down, the semaphore for going down is grabbed, as well as the semaphore for going up. A counter for counting cars going down is then incremented, and the semaphore for going down is released. This allows other cars to go down, while preventing cars from going up. When leaving the alley, cars just decrement the counter for how many is going down, and if it is the last one in the alley (i.e cars going down = 0), it releases the semaphore for going up, which was grabbed by the first car to go down.

The procedure for going up is the same but opposite.

According to the assignment (and our intuition of course), we are a bit suspicious about the solution, and we will therefore investigate with Spin:

An issue is that 2 cars might both be at the mouth of the alley, one in the top and one on the bottom. Both cars take their individual semaphores using their P() commands, which will leave both of them eternally stuck, when they begin waiting for one another. This can be seen from the Promela code we've provided, if there is no assertion check. The program reaches an invalid end state, since the program deadlocks and is stuck. This is due to multiple cars entering their P() commands on lines 38 and 25 at the same time.

Process	Statement	Car(2):tem	Car(2):tem	Car(3):tem	Car(3):tem	downSem	up	upSem	upSpin
3 Car	82 up==0	0	0	0	0	0	0	1	0
3 Car	83 downSem = 1	0	0	0	0	0	0	1	0
3 Car	34 else	0	0	0	0	1	0	1	0
3 Car	35 upSem==1	0	0	0	0	1	0	1	0
1 Car	22 downSem = 0	0	0	0	0	1	0	1	0
2 Car	34 else	0	0	0	0	0	0	1	0
3 Car	35 upSem = 0	0	0	0	0	0	0	1	0
3 Car	37 up==0	0	0	0	0	0	0	0	0
1 Car	24 down==0	0	0	0	0	0	0	0	0

spin: trail ends after 170 steps

In the Promela code, we've introduced some independent variables, upSpin and downSpin, to create an assertion statement which must be true for the program to not deadlock. They mimic our variables which count how many cars are currently going up or down, however upSpin and downSpin are incremented atomically. If cars are going both directions, meaning both variables are positive, then we have a deadlock.

Thus our assertion statement, and our alley safety property, is as follows:

$(upSpin <= \frac{N}{2} \wedge downSpin = 0) \vee (downSpin <= \frac{N}{2} \wedge upSpin = 0)$

This can be violated, as the variables counting the amount of cars going up or down are not atomically incremented in the program. This means We can have conflicts with cars going up or down. It can be seen in the following jSpin picture.

```
Process Statement          Car(1):tem Car(2):tem Car(2):tem Car(3):tem Car(3):tem down       downSem    up
upSem      upSpin
3 Car   38  downSem = 0    0          0          0          0          0          1          1          0
1          0
3 Car   41  temp2 = up     0          0          0          0          0          1          0          0
1          0
3 Car   42  up = (temp2+1) 0          0          0          0          0          1          0          0
1          0
3 Car   44  upSem = 1      0          0          0          0          0          1          0          1
1          0
3 Car   51  else           0          0          0          0          0          1          0          1
1          0
3 Car   52  upSpin = (upSp 0          0          0          0          0          1          0          1
1          0
3 Car   56  assert((((upSp 0          0          0          0          0          1          0          1
1          1
3 Car   61  else           0          0          0          0          0          1          0          1
1          1
3 Car   62  upSpin = (upSp 0          0          0          0          0          1          0          1
1          1
3 Car   77  else           0          0          0          0          0          1          0          1
1          0
3 Car   79  temp4 = up     0          0          0          0          0          1          0          1
1          0
3 Car   80  temp4 = (temp4 0          0          0          0          1          1          0          1
1          0
```

Figure 1: Assertion violation

Problem 5+

When making the coarse Promela model where the enter and leave operations are defined as larger atomic actions, our actions use an *up* as well as a *down* variable to count the amount of cars going either up or down. As such, we can have a very similar alley safety property to previous exercises, which looks like the following and ensures no deadlocks:

$$(down = \frac{N}{2} \wedge up == 0) \vee (up <= \frac{N}{2} \wedge down == 0)$$

When implementing the final model in the Man2baton file, we've made slight changes from Andrews 4.4.3 to ensure greater clarity in the code. When one of our processes passes the baton to another delayed process, the delayed process is then responsible for counting down the amount of delayed processes, instead of the signaling process.

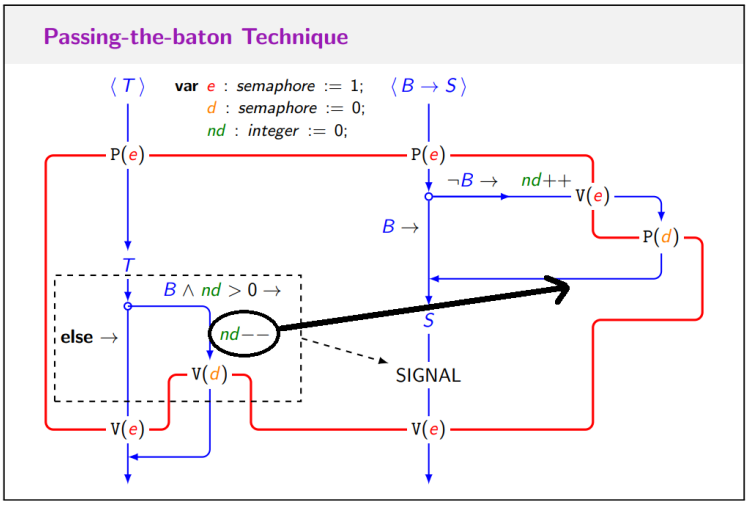

Figure 2: The delayed process is responsible for counting down the number of delayed processes.

Finally we need to verify the *split binary semaphore* principle. Our control semaphores are *upSem*, *downSem* and *enterSem*, which are all binary. We verify the principle by asserting that the sum of our semaphores is less than or equal to 1. We do so at the same time as our other assertion, which is in the middle of the critical section.

$$downSem + upSem + enterSem <= 1$$

Conclusion

In this assignment, we have helped the local playground protects its new electric moon-cars, as well as the kids driving them. In the old cars, they kept bumping into each other, which can be dangerous at high speeds, they also kept crashing into each other when going down the alley behind the hut from both ways.

In our solution, we started off by making sure the new electric moon-cars could not bump into each other. This we did by dividing the playground into sections, and not allowing more than one car in each section at any one time.

This then lead to deadlock in the alley, since cars could no longer RAM through each other, to pass through. Therefore, a synchronization solutions was needed. Two was provided for analysis, where one was only allowing one car in the alley at a time, where the alley was a single critical region. The other allowed for multiple cars going in one direction, but was error prone due to implementation semantics, causing either deadlock for going into the alley or breaks the safety property and allows cars going in both directions at the same time. This was all modeled in spin to reach a final analysis.

Finally we had to implement our own solution to the synchronization problem. We chose to use the principle of passing the baton, with an extended read/writer principle, except that we allow for multiple cars in both directions, and not just one. This is equivalent to having two groups of readers, which exclude each other.

In the end we succeeded with making sure the cars cannot bump into each other, and that they wait for the alley to be clear before driving through it, while still allowing for multiple cars going in the same direction at the same time, keeping the kids happy.