02158 - Concurrent Programming

DTU - Technical University of Denmark

Date of submission: Okt. 13, 2021

Assignment 2

Daniel F. Hauge (s201186)

Max Thrane Nielsen (s202785)

Abstract

1 Problem 1: A fair critical region

In order to guarantee fairness for all process, the control for accessing the critical region is centralized in a single conductor processor.
The means of synchronization between the process is achieved by setting flags to signal a request to enter the critical region by any process and the approval to a request by the conductor.
We make use of two boolean request/approval arrays storing the flags of every process where each index in an array corresponds to the _pid of a process in the program.
A process will set its request flag to true upon wanting to enter the critical region and will be busy waiting for its approval flag to enter.
A conductor will either be monitoring the elements in the request array in a loop or be busy waiting for a process to leave the critical region.
When the conductor sees a request to enter it assigns the approval flag for the corresponding process to be true and sets the process request flag to false. We have implemented this operation as one atomic operation to avoid race conditions on the writing to the request flag.
The solution adheres to the principle of resetting your own flag, st. the process that gained access by its approval flag being set to true, will also reset the flag after it leaves the critical region, and similarly will the conductor reset the request flag for a process.
The implementation is a round robin style fairness as the conductor will continue the loop from where it stopped thereby responding to the next request, making sure that all process requests are handled in the same order they have been requested in.
We removed the while loop in the critical region to prevent starvation.

2 Problem 2. Avoiding Bumping

In order to avoid cars bumping, each car should only move to a tile whenever it is free, and if occupied then wait until free. In other words, a tile is a resource which only allows one actor to access it at a time. The solution is achieved by letting each tile be a critical region where only one car may occupy it at a time. Tiles becomes a critical region by associating a semaphore lock to it, such that when entering a car will request the corresponding semaphore and release it when leaving. Cars that request to enter a tile which semaphore is locked by another car, will be delayed until the occupying car releases the lock. Using semaphores to achieve mutual exclusion to each tile eliminates car bumping. It is implemented in java by an array of arrays of semaphores, where indices correspond to column and row of the tiles. See Field.cs

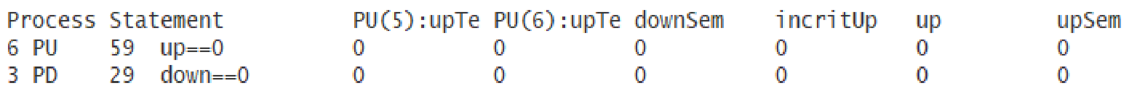
Although eliminating bumping is nice, using semaphores to synchronize also introduce potential issues of safety, such as deadlocks. Example: If a car on tile X is requesting a tile which is occupied by a car that is waiting for tile X to be free.

3 Problem 3: Analyzing an Alley Synchronization Proposal

In the MonoAlley class a critical region has been constructed around the alley by the use of a semaphore, only allowing one car to enter at a time. This prevents dead lock but it is a very conservative approach as every other car has to wait even though there could be multiple cars driving in the same direction in the alley at the same time.

The MultiAlley solution provisions all cars driving in the same direction access to the alley at once, thus being more efficient than MonoAlley. The caveat is that the formation of the cars in one direction could potentially result in a live lock for some process and maybe even a dead

lock.
By inspecting the java implementation it is clear that there is an interleaving where two cars that drive in opposite direction are able to both acquire the control to drive in the alley before either try to acquire the opposite cars lock. This leads to both of them being waiting for the opposite car to release the lock, conventionally called a dead lock.
Reproducing the model from MultiAlley as faith full as possible, we can detect possible problems. Even if the synchronization looks to be working eventually some process could be in starvation. When checking the model in JSpin it was possible to produce the invalid end state failure by the spin verifier. Considering the JSpin guided trail, we can observe a deadlock. Last two lines in the guide trail of the error:



```
Process Statement          PU(5):upTe PU(6):upTe downSem    incritUp   up         upSem
6 PU    59  up==0          0          0          0          0          0          0
3 PD    29  down==0        0          0          0          0          0          0
```

Figure 1: Guide trail of the invalid end state error.

We observe that the values of the two semaphores means they have both been acquired and that the up and down count are also 0 thus both process are locked waiting for the opposite process to release its lock.
The assertion error was not raised, which we could reason about it that since the process deadlock when both acquire the lock to access the alley.

4 Problem 5+: Baton-solution

4.1 Coarse grained model

The alley problem is a problem of a shared resources which cannot be accessed in different ways by multiple actors. However, accessing the resources in the same manner by multiple actors will still ensure execution progress. In the specific case of the alley problem, there are two different ways of accessing the resources/alley, which is accessing from top to bottom and bottom to top. Let the ways of accessing the alley be P_u and P_d, and represent a predicate for whether the resources is being accessed at a given time by this way.

In order to achieve a deadlock-free system, the model has to guarantee that different ways of accessing the alley simultaneously is impossible. The problem is solved by letting the alley be a critical region, which only allows either P_u, P_d or neither to access the region at a given time. This give rise to the following temporal logic:

$$\Box((\neg P_u \land P_d) \lor (P_u \land \neg P_d) \lor (\neg P_u \land \neg P_d)) \implies \Box(\neg(P_u \land P_d))$$

The model can be expressed by two kind of cars which will use either P_u or P_d to cross the alley. Each car will wait until no cars in opposite direction are in the alley, before entering the alley. The car will wait based on a direction control variable namely *up* and *down*, which each car will increment and decrement during entering and exiting of the alley. Evaluating wait condition, increments and decrements has to happen atomically to ensure safety.

Car using: P_d	Car using: P_u
Entry: $\langle await(up == 0); down++; \rangle$	Entry: $\langle await(down == 0); up++; \rangle$
Crit: // Critical region (Alley)	Crit: // Critical region (Alley)
Exit: $\langle down--; \rangle$	Exit: $\langle up--; \rangle$

See Man2coarse.pml for full coarse promella model.

With the model, the predicates can be defined as, being true if one or more cars of a direction are in the critical section:

$$P_x = \sum [Car_x@Crit] > 0$$

In promella, this logic is asserted more explicitly by history variables *incritDown* and *incritUp*. The following promella code is the critical section of processes using P_d:

```
incritDown++;
assert(incritDown <= NDown);
assert(incritUp == 0);
incritDown--;
```

Using jSpin, the coarse model is verified to be safe, ie. free of deadlocks.

N.B amount of cars is lowered to 2 up and 3 down cars, as it would otherwise be difficult to verify with exceedingly large state-spaces.

4.2 Passing the baton technique

Implementing the model can be done in serveral ways, such as the inefficient busy-waiting technique. However a more efficient and cool way is "Passing the baton". Passing-the-baton puts waiting threads to sleep such that it will not waste clock cycles in the CPU evaluating the condition which might not be satisfied yet. From the coarse model, the new model using passing-the-baton is systematically implemented using semaphores and control varibles as described in [Andres 4.4.3]. See Man2baton.pml for the baton promella model. The baton model is verified to be safe in promella.

Where as the coarse model would not make a very faithfull java implementation, the derived baton model can make a very faithfull java implementation. See the java implementation in BatonAlley.java.

4.3 Split binary semaphore principle

In order for the baton model to work as intended, it has to satisfy the invariant that only one baton exists. However, the model incorporate multiple semaphore's, which consequently means the model has to ensure that the semaphore's are technically working as a single semaphore/baton. Multiple semaphore's working as one is the split binary semaphore principle. The semaphores in the model are as follows:

- eSem : The semaphore which guards start of entry and exit.
- downSem : The semaphore where procceses/cars wanting to drive down through the alley wait to be passed the baton.
- upSem : The semaphore where procceses/cars wanting to drive up through the alley wait to be passed the baton.

In the model, this invariant can be expressed as: The sum of all semaphores is always 1 or 0, corresponding to a process holding the baton, or the baton is available for pickup. This can be expressed in LTL:

$$\Box(eSem + downSem + upSem <= 1)$$

This invarient can be verified by jSpin with the following:

```
ltl split_property{[](eSem + downSem + upSem <= 1)}
```

Verifying with jSpin reveals the property is satisfied, as all reachable states was searched and it was impossible to find a state where the property do not hold.

5 Problem 6: Alley Liveness

5.1 Resolution

At least one car is always able to move, akin to resolution which is: It should always be the case that at least one of the processes will make progress.

$$\exists i : P_i@entry \rightsquigarrow \exists j : P_j@exit$$

Which can be written in promella friendly manner:

$$\Box(P_i@entry||P_{i+1}@entry||\ldots) \implies \diamond(P_j@exit||P_j + 1@exit||\ldots)$$

This is used and written for the model with the following ltl in promella:

```
ltl res1 { [] (
    Pu[up1]@entry ||
    Pu[up2]@entry ||
    Pd[down1]@entry ||
    Pd[down2]@entry ||
    Pd[down3]@entry)
    -> <> (
    Pu[up1]@crit ||
    Pu[up2]@crit ||
    Pd[down1]@crit ||
    Pd[down2]@crit ||
    Pd[down3]@crit) )}
```

Verifying with jSpin reveals that the property is satisfied, as with exhaustively visiting every state, it cannot get to a state where the property does not hold. Therefor it is impossible to get to a state where the invariant no longer hold.

5.2 Obligingness

A car wanting to go down through an unoccupied alley should be able to proceed without delay. This is akin to obligingness, which is satisfied when a process without hindrance in the form of other processes blocking critical regions / resources will make progress without delay.

$$P_i@entry \wedge (\forall j \neq i : \Box(\neg(P_j@crit))) \rightsquigarrow P_i@crit$$

This is used and written for the model with the following ltl in promella:

```
ltl obl { [] ( ((Pd[down1]@entry) &&
[] !(   Pu[up1]@crit ||
        Pu[up2]@crit ||
        Pd[down2]@crit ||
        Pd[down3]@crit)
) -> <> (Pd[down1]@crit) )}
```

Verifying with jSpin reveals that the property is satisfied. It is impossible to get to a state where the property does not hold.

5.3 Fairness

A car wanting to go up through the alley will eventually be able enter the alley, akin to fairness, which is satisfied when a process will succeed eventually.

$$P_i@entry \leadsto P_i@crit$$

Fairness for a process driving up through the alley can be expressed in promella by the following ltl:

```
ltl fair1 { [] ( (Pu[up1]@entry) -> <>  (Pu[up1]@crit) ) }
```

Verifying in jSpin with disabled weak fairness reveals the lack of fairness, as there exists a cycle which can escape progress for the up1 process. It is revealed by the output as seen below:

```
pan:1: acceptance cycle (at depth 326)
pan: wrote Man2live.pml.trail
```

The unfairness exists because of the following:

If down cars can keep entering the alley before all the remaining down cars leave, down cars can keep occupying the alley forever, essentially starving up cars from ever entering the alley. The example also apply vice versa.

6 Conclusion

A single semaphore is easy to work with, but with more demanding synchronization schemes, the implementation gets very complicated. Semaphores are super efficient and a lot more elegant and waste less than busy-wait. Semaphores will put the thread to sleep when it is waiting during a P(e) call, thus not wasting CPU cycles, therefor contributing to the survival of our planet. Although demanding synchronization schemes make implementations complicated, semaphores is a very convenient mechanism for controlling resources. As semaphores make it easier to control resources, it can be utilized to implement larger atomic actions which is required to circumvent some concurrency safety issues. Possible violation of the safety or liveliness properties of a program during execution can be very hard to reason about, especially if the concurrency schema is very complex. When assessing if a concurrent program might fail to synchronize properly, it is very natural to think about never claims or certain assertions that should hold during the program execution. A verification tool like spin therefor comes very handy as it is possible to make an exhaustive search of the inter-leavings of the program whilst checking if the defined liveliness or safety properties holds in any state of the program. Being sure that a program will not fail during critical moments can be very important. Ensuring correct behavior forever and in all circumstances can vary in importance, but ensuring correct behavior is never a bad thing.