
# Concurrent Programming Assignment 2

**Authors**

Anton Stockmarr - s164170
Andreas Work - s174128
Lucas Kolding - s213556

October 13, 2021

# Part 1: BUSY WAITING

## Problem 1 - A fair critical region

In our solution two flags are used for every process. One called ENTER[I], indicating that process $i$ wants to enter the critical region, and one called OK[I], indication that process $i$ has been let into the critical region.

We have achieved a fair solution by introducing a for loop for the coordinator, which iteratively checks if there is a process that wants to enter the critical region. Once it has found a ready process it will set the OK flag, and use busy waiting until that process has exited the critical region. We ensure mutual exclusion by having the process that exits the loop set the OK flag to FALSE, confirming that a new entry flag can be set. Similarly, the coordinator reads the ENTER flags, and when it finds one that is true, it immediately resets that flag before letting the process enter.

This is a fair solution, which is also validated by the *ltl* constraint in the promela program. It is fair because the coordinator iterates through all processes in a specific order. If a process wants to enter the critical region, the coordinator will eventually get to that process and allow it in. If one process wants to go in again, it will have to wait until all the other processes have had a chance. However, since this solution is based on busy waiting, it might not be fair if the coordinator is on the same core as the other processes, because it actively needs to check the enter condition to allow any processes in.

```
1  /* DTU Course 02158 Concurrent Programming
2   *   Lab 2
3   *   spin5.pml
4   *     Skeleton PROMELA model of mutual exlusion by coordinator
5   */
6
7  #define N 5
8
9  bool enter[N];  /* Request to enter flags */
10 bool ok[N];     /* Entry granted flags    */
11
12 int incrit = 0; /* For easy statement of mutual exlusion */
13
14 /*
15  * Below it is utilised that the first N process instances will get
16  * pids from 0 to (N-1).  Therefore, the pid can be directly used as
17  * an index in the flag arrays.
18  */
19
20 active [N] proctype P()
21 {
22         do
23         ::      /* First statement is a dummy to allow a label at start */
```

```
24                  skip;
25
26  entry:
27                  enter[_pid] = true;
28                  /*await*/ ok[_pid] ->
29
30  crit:   /* Critical section */
31                  incrit++;
32                  assert(incrit == 1);
33                  incrit--;
34
35  exit:
36                  ok[_pid] = false;
37
38                  /* Non-critical setion (may or may not terminate) */
39                  do :: true -> skip :: break od
40
41          od;
42  }
43
44  active proctype Coordinator()
45  {
46          int i;
47          do
48          ::
49                  for (i : 0 .. N-1) {
50                  if
51                  :: enter[i]  -> enter[i]  = false; ok[i] = true; /*await*/ !
                       ok[i] -> /* guard - else*/
52                  :: else -> skip;
53                  fi
54
55                  }
56          od
57  }
58
59  ltl fairness_1 { [] ( P[0]@entry -> <> P[0]@crit ) }
```

# Part 2: SEMAPHORES

### Problem 2 - Avoid Bumping

To avoid bumping each position on the board needs a semaphore. This is achieved by mapping all positions to a semaphore. All semaphores have initial value 1. To enter a position, one must call P() on that position. When one wants to leave, one must call V(). The result is that two or more cars can never be at the same position at the same time. To do that those cars would need to call $P(S)$ before any of them called $V(S)$, which is not possible when the semaphores have value 1.

```
1  import java.util.HashMap;
2  import java.util.Map;
3
4  // Solution to Problem 2: Avoid Bumping
5  public class Field {
6
7      private final Map<Pos, Semaphore> positionStatus;
8
9      public Field() {
10         positionStatus = new HashMap<>();
11         for (int i=0; i <= Layout.COLS; i++){
12             for (int j=0; j<= Layout.ROWS; j++) {
13                 Pos pos = new Pos(i,j);
14                 Semaphore mutex = new Semaphore(1);
15                 positionStatus.put(pos, mutex);
16             }
17         }
18     }
19
20     /* Block until car no. may safely enter tile at pos */
21     public void enter(int no, Pos pos) throws InterruptedException {
22         Semaphore mutex = positionStatus.get(pos);
23         mutex.P();
24     }
25
26     /* Release tile at position pos */
27     public void leave(Pos pos) {
28         Semaphore mutex = positionStatus.get(pos);
29         mutex.V();
30     }
31 }
```

### Problem 3 - Analyzing an Alley Synchronization Proposal

MonoAlley implements a semaphore for the entire alley, stating that only one car may be inside the alley. The enter protocol is to execute $P(S)$, and to leave is $V(S)$. One of the cars trying to enter the alley will always be allowed in as soon as the alley is empty. In this system all cars eventually leave the alley, so deadlocks cannot occur. Whether the solution is fair or not depends on the implementation of the semaphore. If it is based on a FIFO queue it would be fair, since all cars will eventually be allowed in. Otherwise, it might not be fair.
MultiAlley implements a semaphore for each direction of the alley (up, down). It uses variables to store the number of cars going up and down in the alley. If one of these integers goes from 0 to 1, the semaphore for that direction executes $P(S)$. If it goes from 1 to 0, the semaphore executes $V(S)$. Thus, any number of cars can enter the alley as long as they go in the same direction. This solution is not fair. If there are always cars going in one direction in the alley, the other cars might never be allowed in. As it turns out, the solution does not satisfy neither the liveness property nor the safety property.

Below is the Promela code corresponding to the multiAlley program. Two different kind of processes are used, one for cars going up and one for cars going down. They each have their own integer for checking the safety property, namely *upIncrit* and *downIncrit*. If an UP process is in the critical region, then *downIncrit* must be 0. Reversely, if a DOWN process is in a critical region, then *upIncrit* must be 0. These two variables are historic variables, used to verify the safety property. They are incremented and decremented atomically to avoid race conditions. The model still respects the behavior of the java program.

```
1  #define N 2
2
3  #define V(S) atomic{S = S + 1}
4  #define P(S) atomic{S > 0 -> S = S - 1}
5
6  int upSem = 1;
7  int downSem = 1;
8
9  int up = 0;
10 int down = 0;
11
12 int upIncrit = 0;
13 int downIncrit = 0;
14
15
16 /* Up processes */
17 active [N] proctype UP()
18 {
19         do
20         ::      /* First statement is a dummy to allow a label at start */
21                 skip;
```

```
22
23  enter:
24                  P(upSem)
25                  if
26                          :: up == 0 -> P(downSem)
27                          :: else -> skip
28                  fi;
29                  up = up + 1;
30                  V(upSem);
31
32  crit:   /* Critical section */
33                  upIncrit++;
34                  assert(downIncrit == 0);
35                  upIncrit--;
36
37  leave:
38                  up = up - 1;
39                  if
40                          :: up == 0 -> V(downSem)
41                          :: else -> skip
42                  fi;
43
44                  /* Non-critical setion (may or may not terminate) */
45                  do :: true -> skip :: break od
46
47          od;
48  }
49
50  /* Down processes */
51  active [N] proctype DOWN()
52  {
53          do
54          ::      /* First statement is a dummy to allow a label at start */
55                  skip;
56
57  enter:
58                  P(downSem)
59                  if
60                          :: down == 0 -> P(upSem)
61                          :: else -> skip
62                  fi;
63                  down = down + 1;
64                  V(downSem);
65
```

```
66  crit:   /* Critical section */
67                  downIncrit++;
68                  assert(upIncrit == 0);
69                  downIncrit--;
70
71  leave:
72                  down = down - 1;
73                  if
74                          :: down == 0 -> V(upSem)
75                          :: else -> skip
76                  fi;
77
78                  /* Non-critical setion (may or may not terminate) */
79                  do :: true -> skip :: break od
80
81          od;
82  }
```

When analyzing the error trail in SPIN, we find that at a certain point, we will experience a deadlock due to the entry protocol allowing both the Up and Down process to enter at the same time. When we traced the error, we could see that if one process e.g. down would $P(downSem)$ and before it could enter the if statement and set $P(upSem)$, the Up process could enter the entry protocol and $P(upSem)$, before the down processes, which meant that the two processes could not proceed and would therefore be stuck in the entry protocol.

As for the safety violation, we traced that we had two processes trying to call $V(upSem)$ or $V(downSem)$ at the same time, and when doing so, both processes would be allowed access to the critical section. As we see in our trace log - Process down is in line 63 and 72, trying to increment and decrement down at the same time, which leads to them both calling $V(upSem)$, making upSem 2. So in short, the safety violation happens because the increment and decrements of *down* and *up* are not atomic.

### Problem 5 - Alley Safety

The solution in MultiAlley had two issues, one with deadlocks, and one with safety violations. In this section, each of these two issues are solved to obtain a solution that ensures liveness and safety (but not fairness). The code is seen below.

In order to solve the deadlocks, the new program will ensure that only one process can be in the enter protocol at a time. This is done by creating a $enterSem$ semaphore with value 1. The enter protocols for both UP and DOWN processes start with executing $P(enterSem)$ and end with executing $V(enterSem)$. The changes are in line 27, line 35, line 66 and line 74. Now it can no longer happen that two processes are waiting for each other in the enter protocol. A process can only be blocked in the enter protocol if it is waiting for one or more processes in the critical region.

In order to solve the safety violation, the new program uses existing semaphores in the leave protocol to ensure that there are no race conditions when the processes try to decrement the up and down counters. Specifically, the UP processes execute $P(upSem)$ at the start of the leave protocol and $V(upSem)$ at the end of the leave protocol. The DOWN processes do the same for $downSem$. This way only process can evaluate the if statement at a time, and no race conditions occur.

```
1  #define N 2
2
3  #define V(S) atomic{S = S + 1}
4  #define P(S) atomic{S > 0 -> S = S - 1}
5
6
7
8  int upSem = 1;
9  int downSem = 1;
10 int enterSem = 1;
11
12 int up = 0;
13 int down = 0;
14
15 int upIncrit = 0;
16 int downIncrit = 0;
17
18
19 /* Up processes */
20 active [N] proctype UP()
21 {
22         do
23         ::      /* First statement is a dummy to allow a label at start */
24                 skip;
25
26 enter:
27                 P(enterSem);
28                 P(upSem);
29                 if
30                         :: up == 0 -> P(downSem)
31                         :: else -> skip
32                 fi;
33                 up = up + 1;
34                 V(upSem);
35                 V(enterSem)
36
37 crit:   /* Critical section */
```

```
38                  upIncrit++;
39                  assert(downIncrit == 0);
40                  upIncrit--;
41
42  leave:
43                  P(upSem);
44                  up = up - 1;
45                  if
46                          :: up == 0 -> V(downSem)
47                          :: else -> skip
48                  fi;
49                  V(upSem);
50
51                  /* Non-critical setion (may or may not terminate) */
52                  do :: true -> skip :: break od
53
54          od;
55  }
56
57
58  /* Down processes */
59  active [N] proctype DOWN()
60  {
61          do
62          ::      /* First statement is a dummy to allow a label at start */
63                  skip;
64
65  enter:
66                  P(enterSem);
67                  P(downSem);
68                  if
69                          :: down == 0 -> P(upSem)
70                          :: else -> skip
71                  fi;
72                  down = down + 1;
73                  V(downSem);
74                  V(enterSem);
75
76  crit:   /* Critical section */
77                  downIncrit++;
78                  assert(upIncrit == 0);
79                  downIncrit--;
80
81  leave:
```

```
82                  P(downSem);
83                  down = down - 1;
84                  if
85                          :: down == 0 -> V(upSem)
86                          :: else -> skip
87                  fi;
88                  V(downSem);
89
90                  /* Non-critical setion (may or may not terminate) */
91                  do :: true -> skip :: break od
92
93          od;
94  }
```

### Problem 5+ - Baton Solution

We started by implementing this solution in a broader and more coarse SPIN file. In this version we have defined both the Enter and Leave protocol as large atomic actions. Inside these actions, we have defined several counters, as checkmarks. We have implemented checkmarks for entering and leaving the Enter/leave protocol. when inside the entry protocol, we will assert inEnter == 1. We also increment a counter of total cars, when the cars enter the entry protocol, as well as a counter, which keeps track of when a car is entering and leaving the critical section.

As for the leave protocol we start by decrementing carsInCrit, as they have now left the critical section and then we increment inLeave, as they are now in the leave protocol. Then we V either down or upSem allowing for cars going opposite and then we decrement inLeave, as we leave the protocol.

```
1   #define N 2
2
3   #define V(S) atomic{S = S + 1}
4   #define P(S) atomic{S > 0 -> S = S - 1}
5
6   int upSem = 1;
7   int downSem = 1;
8
9   int up = 0;
10  int down = 0;
11
12  int upIncrit = 0;
13  int downIncrit = 0;
14  int inEnter = 0;
```

```
15 int inLeave = 0;
16 int carsInCrit = 0;
17 int totalCars = 0;
18
19 /* Up processes */
20 active [N] proctype UP()
21 {
22         do
23         ::      /* First statement is a dummy to allow a label at start */
24                 skip;
25
26 enter:
27                 atomic{
28                 P(upSem);
29                 inEnter++;
30                 assert(inEnter == 1);
31                 totalCars++;
32                 if
33                         :: up == 0 -> P(downSem)
34                         :: else -> skip
35                 fi;
36                 up = up + 1;
37                 V(upSem);
38                 carsInCrit++;
39                 inEnter--;
40                 }
41
42 crit:   /* Critical section */
43                 upIncrit++;
44                 assert(downIncrit == 0);
45                 upIncrit--;
46
47 leave:
48                 atomic{
49                 carsInCrit--;
50                 inLeave++;
51                 assert(inLeave == 1);
52                 up = up - 1;
53                 if
54                         :: up == 0 -> V(downSem)
55                         :: else -> skip
56                 fi;
57                 inLeave--;
58                 }
```

```
59
60                  /* Non-critical setion (may or may not terminate) */
61                  do :: true -> skip :: break od
62
63          od;
64  }
65
66
67  /* Down processes */
68  active [N] proctype DOWN()
69  {
70          do
71          ::      /* First statement is a dummy to allow a label at start */
72                  skip;
73
74  enter:
75                  atomic{
76                  P(downSem);
77                  inEnter++;
78                  assert(inEnter == 1);
79                  totalCars++;
80                  if
81                          :: down == 0 -> P(upSem)
82                          :: else -> skip
83                  fi;
84                  down = down + 1;
85                  V(downSem);
86                  carsInCrit++;
87                  inEnter--;
88                  }
89
90  crit:   /* Critical section */
91                  downIncrit++;
92                  assert(upIncrit == 0);
93                  downIncrit--;
94
95  leave:
96                  atomic{
97                  carsInCrit--;
98                  inLeave++;
99                  assert(inLeave == 1);
100                 down = down - 1;
101                 if
102                         :: down == 0 -> V(upSem)
```

```
103                         :: else -> skip
104                 fi;
105                 inLeave--;
106                 }
107
108                 /* Non-critical setion (may or may not terminate) */
109                 do :: true -> skip :: break od
110
111         od;
112 }
```

We then implemented the coarse atomic actions using the passing-baton method. We added some delay markers *delayedUp* and *delayedDown*. When we enter the entry protocol, we $P(enterSem)$ and then head into the if statement. Here it is checked if up or down is larger than 0. If it is we will delay our process and exit the entry protocol, while still setting $P(upSem)$, which means that it can not be used by another process while waiting for entry. After gaining entry the delay counter is decremented and the upSem semaphore is released by $V(upSem)$.

As for the leave protocol, we again use our enterSem semaphore to ensure that no other processes will interfere. Inside our if statement, we check two parameters: if up is equal to 0 and if delayedDown is less than 0. If the parameters are true, we will pass the baton to the delayed process. otherwise we will exit the leave protocol. This problem is very similar to the reader-writer problem described in the book. The difference is that the reader-writer problem is restricted to have only one writer in the critical region at a time, where this up-down problem can have an arbitrary number of ups or downs at a time.

The solution uses three semaphores, *upSem*, *downSem* and *enterSem*. Only one of these semaphores are 1 at the time in this solution. This is what is called split binary semaphores. Mathematically, a set of semaphores $S$ are split binary semaphores if and only if

$$\sum_{s \in S} s \leq 1 \tag{1}$$

To prove this property for our solution, the promela model includes an invariant, saying that the sum of the three semaphores can never be more than 1. This invariant holds for the program, proving that *upSem*, *downSem* and *enterSem* are split binary semaphores.

```
1 #define N 2
2
3 #define V(S) atomic{S = S + 1}
4 #define P(S) atomic{S > 0 -> S = S - 1}
5
6 int upSem = 0;
7 int downSem = 0;
8 int enterSem = 1;
```

```
9
10  int up = 0;
11  int down = 0;
12  int delayedUp = 0;
13  int delayedDown = 0;
14
15  int upIncrit = 0;
16  int downIncrit = 0;
17
18  /* Up processes */
19  active [N] proctype UP()
20  {
21          do
22          ::      /* First statement is a dummy to allow a label at start */
23                  skip;
24
25  enter:
26                  P(enterSem);
27                  if
28                          :: down > 0 -> delayedUp++; V(enterSem); P(upSem)
29                          :: else -> skip
30                  fi;
31                  up = up + 1;
32                  if
33                          :: delayedUp > 0 -> delayedUp--; V(upSem)
34                          :: else -> V(enterSem)
35                  fi;
36
37  crit:   /* Critical section */
38                  upIncrit++;
39                  assert(downIncrit == 0);
40                  upIncrit--;
41
42  leave:
43                  P(enterSem);
44                  up = up - 1;
45                  if
46                          :: up == 0  -> if :: delayedDown > 0 -> delayedDown
                               --; V(downSem)
47                                                  :: else -> V(enterSem) fi;
48                          :: else -> V(enterSem)
49                  fi;
50
51                  /* Non-critical setion (may or may not terminate) */
```

```
52                  do :: true -> skip :: break od
53
54          od;
55  }
56
57
58  /* Down processes */
59  active [N] proctype DOWN()
60  {
61          do
62          ::      /* First statement is a dummy to allow a label at start */
63                  skip;
64
65  enter:
66                  P(enterSem);
67                  if
68                          :: up > 0 -> delayedDown++; V(enterSem); P(downSem)
69                          :: else -> skip
70                  fi;
71                  down = down + 1;
72                  if
73                          :: delayedDown > 0 -> delayedDown--; V(downSem)
74                          :: else -> V(enterSem)
75                  fi;
76
77  crit:   /* Critical section */
78                  downIncrit++;
79                  assert(upIncrit == 0);
80                  downIncrit--;
81
82  leave:
83                  P(enterSem);
84                  down = down - 1;
85                  if
86                          :: down == 0  -> if :: delayedUp > 0 -> delayedUp--;
                                V(upSem)
87                                          :: else -> V(enterSem) fi;
88                          :: else -> V(enterSem)
89                  fi;
90
91                  /* Non-critical setion (may or may not terminate) */
92                  do :: true -> skip :: break od
93
94          od;
```

```
95 }
96
97
98 ltl binarySplit_1 { [] (0 <= upSem + downSem + enterSem <= 1)}
```

## Conclusion

In this assignment different ways of solving the mutual exclusion problem have been evaluated. This particular problem, the up/down problem, can be seen as an extension of the reader/writer problem, but where there can be both multiple readers and writers, though not at the same time.

First a simple solution was created for the basic mutual exclusion problem with busy waiting. This implementation was both fair and relatively simple. But busy waiting wastes computing power, and the solution works best if the coordinator is on a different core than the other processes. The following exercises test different solutions to the up/down problem using semaphores. Semaphores simplify the issue of mutual exclusion greatly. They can be used to create arbitrary atomic actions, making the implementation of coarse solutions like the one in exercise 5+ easier to implement. Also, semaphores abstract many implementation details away from the program using them, leaving it to the operation system or executing environment to provide their own implementations. However, this gives the program less control over things like whether or not the solution is fair.

In general, tools like JSpin make it a lot easier to spot race conditions in distributed programs. As this course illustrates, one can rarely rely on intuition to find race conditions. Much software developed today need to deal with parallelism, and race conditions can make this kind of software unstable or unsafe, increasing the need for rigorous ways of proving safety, liveness and fairness properties. JSpin also allows increased insights into possible scenarios, that might otherwise not be discovered. As in problem 3, where we find no errors in the Java program, but discover both deadlocks and safety violations when we analyze using JSPIN.