Solutions for CP Exercises, December 4

1. Solution for CP Exam December 1998, Problem 1

Question 1.1

(a) At entry in cs_1 , $C_1 = 1$ due to the loop condition. In cs_1, C_1 is not changed by P_1. Further, the only assignment to C_1 in P_2 is f_2 which cannot change $C_1 = 1$. Thus, $C_1 = 1$ in cs_1.

(b) *Omitted.*

(c) $I \triangleq C_1 = C_2 \Rightarrow C_1 = 0 \wedge C_2 = 0$

I holds initially since $C_1 = 0 \wedge C_2 = 0$.

The potenially dangerous actions are changes of C_1 and C_2. For P_1 these are:

a_1: After the execution, C_1 and C_2 are different (cf. (b)). Thus I holds.

d_1: If $C_2 \neq 0$ before the execution, C_1 and C_2 are different afterwards. If $C_2 = 0$ before, both are 0 afterwards. In both cases, I holds.

f_1: Before the execution, $C_1 = 0$ cf. H_1. Since C_1 is not changed by this action, C_1 and C_2 differ after the execution, hence I holds.

By symmetry, we see that I is also preserved by all actions in P_2. Together with I holding initially, we conclude that I is an invariant for the program.

(d) Assume that mutual exclusion was violated: $in\ ks_1 \wedge in\ ks_2$

According to (a), we should then have $C_1 = 1 \wedge C_2 = 1$ contradicting I. Thus, mutual exlusion is ensured for this program.

Question 1.2 *(Not required)*

It is assumed that P_1 stays in w_1 for ever:

1. If P_1 stays in w_1 for ever, it follows from H_1 that $C_1 > 0$ must hold forever. Thus, we can assume $\Box in\ w_1$ og $\Box C_1 > 0$ in the following.
2. Due to fair process execution, P_2 will eventually reach e_2 unless it gets stuck somewhere else. Since we assume that it cannot remain in cs_2, it can only get stuck in nc_2 or w_2.
3. When P_2 executes the test in the **if**-statement, we either have $C_1 = 1$ or $C_1 > 1$ (since $\Box C_1 > 0$). In the latter case, P_2 moves to f_2.
4. Due to fair process execution, f_2 will be executed eventually, ant right after this, $C_1 = 1$.
5. If at any time $C_1 = 1$ it will remain so for ever since P_1 is assumed to stay in w_1 and P_2 cannot change C_1 to any other value than 1.
6. When P_2 is in nc_2 we cannot have $C_1 > 1$ according to K_1. Since $\Box C_1 > 0$, it must therefore be 1. Thus, we have $\Box in\ ik_2 \Rightarrow \Box C_1 = 1$.
7. If $\Box C_1 = 1$, P_1 will eventually discover this and leave w_1

8. This contradics the assumption $\Box in\ w_1$.

Question 1.3

(a) Since G_i and H_i now become *local invariants*, these are immediately seen to still hold. Likewise, the argument for I holds and hence the mutual exclusion proof is still valid.

(b) Consider the following program execution:

	C_1	C_2
Initielt	0	0
P_1 executes nc_1, a_1, and enters cs_1	1	0
P_2 executes nc_2, a_2	1	2
P_1 executes d_1, nc_1, a_1	3	2

Both process are now caught in w_1. Since a deadlock can occur between the entry-protocols, the implementation is no longer resolute.

2. Solution for Concurrent Systems Exam December 2001, Problem 1

Question 1.1

(a) I holds initially since $y = 0$.

All three a-actions are potentially dangerous for I:

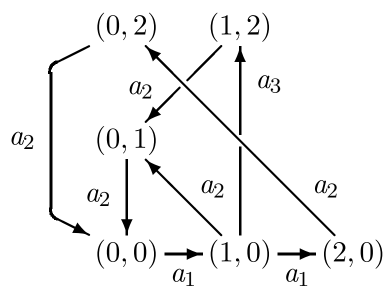
a_1: Is executed only if $y = 0$ and does not change y. Thus $y = 0$ after the action and I holds.

a_2: If $x = 0$ when executed, $y = 0$ after and I holds. If $x > 0$ when executed, $y > 0 \wedge x = 0$ after, i.e. after the action $x \neq y$ and I holds

a_3: After this action we always have $x = 1 \wedge y = 2$, thus I holds after the actions.

Since I holds initially and is preserved by all atomic actions, I is an invariant of the program.

(b) Transition graph:



The initial state is $(0, 0)$. Further there are an a_2 self-loop on the state $(0, 0)$ and an a_3 self-loop on state $(1, 2)$ (not shown).

(c) From the transition graph, we see that the state $(x, y) = (1, 2)$ is reachable and therefore $(x = 0 \vee y = 0)$ is **not** an invariant of the program.

Question 1.2

(a) Given a transition graph, weak fairness ensures that the execution cannot remain forever in a state where there are enabled actions leading to other states. By inspecting the possible exection paths in the transition graph we therefore conclude that any execution must pass through $(x, y) = (0, 0)$ over and over again. Thus, $\Box\Diamond y = 0$ is a property of the program.

(b) Likewise, any execution will have to pass through $(x, y) = (1, 0)$ over and over again. Therefore a_3 will be enabled inifinitely often and by strong fairness we then get that a_3 must be taken infinitely often. We therefore get to $(x, y) = (1, 2)$ infinitely often, i.e. $\Box\Diamond y = 2$ holds for the program.

Question 1.3

(a) I is violated by the interleaving:

$$(0,0) \xrightarrow{b_1} (0,0) \xrightarrow{c_1} (0,0) \xrightarrow{d_1} (1,0) \xrightarrow{b_1} (1,0) \xrightarrow{c_1} (1,0) \xrightarrow{a_2} (0,1) \xrightarrow{d_1} (1,1)$$

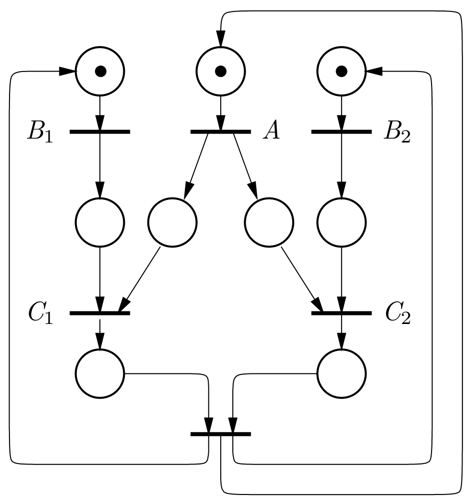
(b) H can be defined as:

$$H \triangleq (x \leq 2) \wedge (\mathit{at}\ c_1 \vee \mathit{at}\ d_1 \Rightarrow x < 2)$$

[The fact that x remains less than 2 at c_1 and d_1 is necessary to ensure that $x \leq 2$ is preserved by d_1.]

3. Solution for Exam June 1991, Problem 2

Question 2.1



Question 2.2

In the solution below, P signals every Q_i after the execution of A. Furthermore P has been appointed a master for the barrier synchronization of all processes after execution of $C_i, \ldots, C_n$.

var $SA[1..n]$: *semaphore* := 0; $\qquad\qquad$ A done

$SC[1..n]$: *semaphore* := 0; — C_i done
$SB[1..n]$: *semaphore* := 0; — OK to start B_i

```
process P =
  repeat
    A;
    for j in 1..n do signal(SA[j]);
    for j in 1..n do wait(SC[j]);
    for j in 1..n do signal(SB[j])
  forever;
```

```
process Q_i[i : 1..n] =
  repeat
    B_i;
    wait(SA[i]);
    C_i;
    signal(SC[i]);
    wait(SB[i])
  forever;
```

[Due to the ending barrier synchronization, it is possible to use *common* semaphores SA and SC instead of $SA[1..n]$ and $SC[1..n]$, but this is **not** true for $SB[1..n]$ since one of the processes may "take an extra coconut" meant for one of the others. Thus, common semaphores should be used only after careful consideration.]

Question 2.3

```
monitor Synch;

  var Ok : boolean := false;           — OK to start (A done)
      Done : integer := 0;             — C's done
      OkStart,
      Alldone : condition;             — Wait for all C_i done

  procedure Done
    Ok := true;
    signal_all(OkStart);
    wait(Alldone);

  procedure Start
    if ¬Ok then wait(OkStart);

  procedure End
    Done := Done + 1;
    if Done < n then wait(Alldone);
                else Done := 0;
                     Ok := false;
                     signal_all(Alldone)

end Synch;
```

[Solution assumes no spurious wake-ups.]