Suggested Solutions for

Written Exam, December 7, 2021

PROBLEM 1

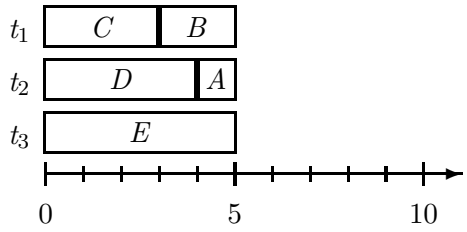
Question 1.1

(a) The speedup cannot exceed the number of processors nor worker threads, hence **2** is the upper limit determined by the number of threads.

(b) Similarly, here **4** is the upper limit determined by the number of processors.

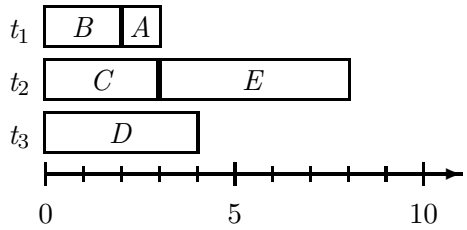
Question 1.2

(a) With the given execution times, a perfect fit is possible:



The resulting exeucition time is **5 seconds** yielding a speedup of $\frac{15}{5} = \mathbf{3}$.

(b) To suffer a speedup less than 2, the execution time would have to be greater than $\frac{15}{2} = 7.5$ seconds. This is the case in the following scenario:



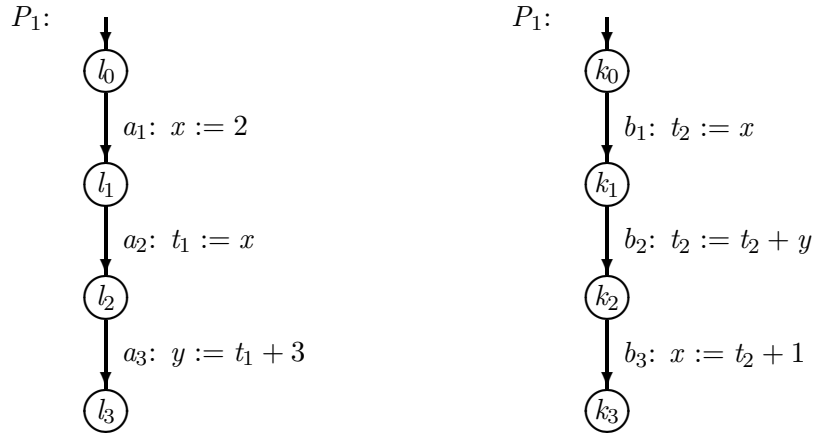
where the execution time becomes 8 seconds..

(c) The tasks could be submitted in reverse alphabetical order: E, D, C, B, and A. Any other sequence ending in B, A is also feasible.

PROBLEM 2

Question 2.1

(a) Transition diagrams:



[Left-to-right evaluation assumed. Location and action labels not required.]

(b) The final values of x can be found by going through the 20 possible interleavings. Alternatively, we may observe that the value is detmined by either a_1 or b_3. In the case of b_3 the value further depends on the ordering of a_1/b_1 and of a_3/b_2. Either way, the possible values are found to be

$$1, 2, 3, 6, 8$$

Question 2.2

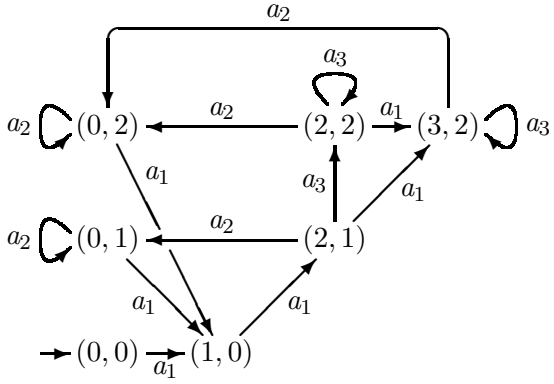
P is preserved by a and c. By a, sinde the guard ensures that $y > 0$. Not by b if y is negative. Byc as x is incremented.

Q is preserved by all three actions. Obviously by a. By b as the zero-ness of y is preserved. Finally the guard of c ensures that $x = 0$ is preserved (if $y \neq 0$).

R is preserved by b and c. Not by a as it may violate $0 < x$. By b as it makes y more positive and by c, as the guard is not satisfied for R.

Question 2.3

(a) Transition graph:



(b) **Assuming weak fairness**

- F **does not** hold. Among the reachable states, the condition $(x + y = 0)$ is satisfied only for $(0, 0)$. However, since this state must be left due to weak fairness and cannot be entered again, F cannot hold.
- G **holds**. The condition $y = 2$ can occur in the states $(0, 2)$, $(2, 2)$, and $(3, 2)$. From $(2, 2)$ and $(2, 3)$ the state $(0, 2)$ must eventually be reached and from there $(1, 0)$ is reached.
- H **does not** hold. States with $y = 2$ may be avoided by the infinite execution

$$(0,0) \xrightarrow{a_1} (1,0) \xrightarrow{a_1} (2,1) \xrightarrow{a_2} (0,1) \xrightarrow{a_1} (1,0) \xrightarrow{a_1} \cdots \qquad (*)$$

  satisfying weak fairness since a_3 is not continously enabled and hence does not have to be executed.
- I **does not** hold. The state $(2, 2)$ is the only reachable state satisfying $(x + y = 4)$. Hence it follows from above.

Assuming strong fairness

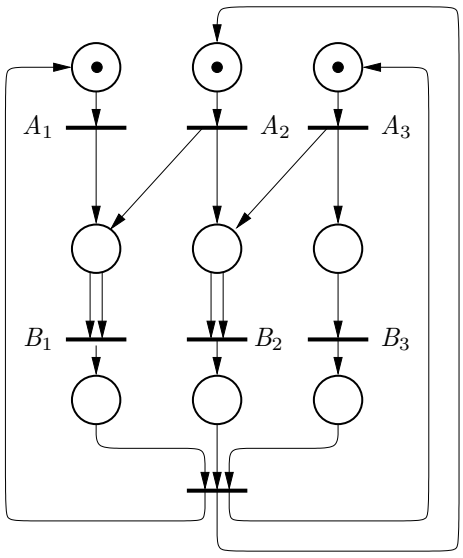
- F **does not** hold. Strong fairness does not change the fact that $(0, 0)$ cannot be reentered, once left.
- G **holds**. Since G holds for weak fairness, it also holds for strong fairness.
- H **holds**. In order to avoid a state with $y = 2$ the execution would have to follow the execution $(*)$ (possibly taking some extra a_2 loops in $(0, 1)$). However, in such an execution, a_3 is infinitely often enabled without being taken thus violating strong fairness. Therefore states satifying $y = 2$ must be visited infinitely often.
- I **does not** hold. The state $(2, 2)$ can be avoided by the infinite execution

$$(0,0) \xrightarrow{a_1} (1,0) \xrightarrow{a_1} (2,1) \xrightarrow{a_1} (3,2) \xrightarrow{a_3} (3,2) \xrightarrow{a_2} (0,2) \xrightarrow{a_1} (1,0) \xrightarrow{a_1} \cdots$$

  This satisfies strong fairness since all tree actions are executed infinitely often.

PROBLEM 3

Question 3.1



Question 3.2

In order to control the ending barrier one of the processes, P_n, is appointed coordinator for the barrier synchronization. Otherwise, the solution reflects the synchronization pattern shown in the Petri Net.

```
var SA[1..n-1] : semaphore;        — A_{i+1} done
    SB[1..n-1] : semaphore;        — B_i done
    SC[1..n-1] : semaphore;        — OK to restart P_i
```

All semaphores are initialized to 0

```
process P[i : 1..n-1];
  repeat
    A_i;
    if i > 1 then V(SA[i-1]);
    P(SA[i]);
    B_i;
    V(SB[i]);
    P(SC[i])
  forever
```

```
process P_n;
  repeat
    A_n;
    V(SA[n-1]);
    B_n;
    for j in 1..n-1 do P(SB[j]);
    for j in 1..n-1 do V(SC[j])
  forever
```

[It is possible to use *common* semaphore SB instead of $SB[1..n]$.]

Question 3.3

```
monitor Synch

  var adone[1..n] : boolean := false;   — A done flags
      done : integer := 0;              — No. of B's done
      TryA : condition;                 — Await some new A done
      Alldone : condition;              — Await all B's done

  procedure doneA(i : integer) {
    adone[i] := true;
    if i > 1 then
      if adone[i-1] then signal_all(TryA)
    if i < n then
      while ¬adone[i+1] do wait(TryA);
  }

  procedure doneB() {
    done := done + 1;
    if done < n then wait(Alldone)
                else { done := 0;
                       for j in 1..n do
                         adone[j] := false;
                       signal_all(Alldone) }
  }

end
```

[The solution uses a covering condition for $TryA$. A more efficient solution could use an array of condition queues to be signalled individually.]

PROBLEM 4

Spørgsmål 4.1

(a) $limit = 500$ means that the packing process has called *fill* and is waiting for the bag to be filled with (at least) 500 grammes of candies. $sum = 215$ says that candy processes with a total of 215 grammes of candies have been allowed to pour and $count = 3$ means that three of these have not yet finished doing so.

(b) As *count* is initialized to 0 and the candy processes always call *start*() before *end*(), for each decrement of *count* in *end*() it will always have been incremented once in *start*(). Hence I follows.

(c) The packing process should wait only if the filling is not yet (completely) finished:

$$I_Q \triangleq waiting(Filled) > 0 \Rightarrow sum < limit \vee count > 0$$

Proof:

- I_Q holds initially as *Filled* is empty.
- If I_Q holds at the beginning of *start*, nothing has changed if the process waits. If the while-loop is passed, *sum* is incremented (possibly violating $sum < limit$), but also *count* is incremented making $count > 0$ true. Hence I_Q is preserved by *start*.
- In *end* the decrement of count may make $count = 0$ but if also $sum \geq limit$ the condition *Filled* will be signalled and as as most one process can be waiting, the queue will be emptied. Hence I_Q is preserved by *end*.
- In *fill*, the clearance of *sum* ensures that $sum < limit$ will hold at the wait. After the wait, the condition queue is empty and I_Q is preserved.

As I_Q holds initially and is preserved by all operations, it is a monitor invariant.

(d) Only the wait in *fill* is vulnerable. It may be protected by the condition found in (c):

```
procedure fill(g : posinteger) {
    ⋮
    while sum < limit ∨ count > 0 do wait(Filled)
    ⋮
}
```

Question 4.2

As the number of candy processes needed cannot be determined before the capacity of the bag is known, one is forced to accept a call of *fill* and then hold the rendezvous until the bag has been filled:

```
module Pack
  op start(w : posinteger);
  op end();
  op fill(g : posinteger);
body
  process Control;
    var sum : integer := 0;
        count : integer := 0;
    repeat
      sum := 0;
      in fill(g : posinteger) →
           while sum < g ∨ count > 0 do
             in start(w : posinteger) and sum < g→ sum := sum + w;
                                                   count := count + 1
             [] end()                            → count := count − 1
             ni;
      ni
    forever;
end Pack;
```

[Unlike the monitor solution, this task will block for calls of *end* until *fill* is called. In the given process system, this does not matter, though.]

Question 4.3

(a) The pouring should be started when the candy processes that have called *start* together have enough candies to fill the current bag. Only the necessary number of candy processes should be started if enough candy is already pending when *fill* is called.

(b) A solution can be obtained by introducing an extra condition queue *Go* holding back the P processes untill they have candies enough for filling the current bag.

```
monitor MixPack;
  var sum, limit, count := 0;
      BagReady, Filled, Go : condition;
  procedure start(w : posinteger) {
    while sum ≥ limit do wait(BagReady);
    sum := sum + w;
    count := count + 1;
    if sum < limit then {signal(BagReady); wait(Go) }
                   else signal_all(Go)
  }
  Prodecures end() and fill() as before
end
```

Alternatively, the amount of candy not yet dispensed can be kept in a variable *pending*. When the bag is ready and enough candy is pending, the pouring starts until the bag capacity is met. Then, *pending* is reduced with the dispensed amount.

```
monitor MixPack;
  var sum, limit, pending, count := 0;
      BagReady, Filled : condition;

  procedure start(w : posinteger) {
    pending := pending + w;
    while sum ≥ limit ∨ pending < limit do wait(BagReady);
    sum := sum + w;
    count := count + 1;
    if sum < limit then signal(BagReady)
                   else pending := pending − sum
  }
  procedure end() {
    count := count − 1;
    if sum ≥ limit ∧ count = 0 then signal(Filled)
  }
  procedure fill(g : posinteger) {
    limit := g;
    sum := 0;
    if pending ≥ g then signal(BagReady);
    wait(Filled);
    limit := 0
  }
end
```

[It is assumed that the candy processes together can provide enough candies to fill any (realistic) bag. If this is not the case, the pouring should be started as soon as all candy processes are present, even if they will not be able to fill the bag.]

(c) Blocking all the calls of *start* until a sufficient amount of candy is ready calls for accessing the parameters of several calls in the invocation queue for *start*. This cannot be expressed by the standard rendezvous construct and hence a server-based solution is **not** immediately feasible.

If the number n of candy processes is fixed (and not too big), a syntactical nesting of n **in** *start*() → ... **ni** constructs may be used though. Alternatively, the nesting may be obtained dynamically by wrapping the **in** construct within a *recursive* procedure.

PROBLEM 5

Question 5.1

(a) One of the remaining two instances can be granted to P_1 which may then finish after which P_2 can finish. Therefore, the situation is *safe*.

(b) The number of available instances is $8 - (r_1 + r_2)$. The number of remaining instances to be claimed P_1 and P_2 is given by $4 - r_1$ and $6 - r_2$ respectively. Finally the situation is safe if just one of the processes can get all its remaining instances:

$$\begin{aligned}(6 - r_2) \leq 8 - (r_1 + r_2) \vee (4 - r_1) \leq 8 - (r_1 + r_2) \quad &\Leftrightarrow \quad 6 \leq 8 - r_1 \vee 4 \leq 8 - r_2 \\ &\Leftrightarrow \quad r_1 \leq 2 \vee r_2 \leq 4\end{aligned}$$

Hence

$$\mathit{Safe} \triangleq r_1 \leq 2 \vee r_2 \leq 4$$

Question 5.2

process *Adm*;
  var r_1, r_2 : *integer* := 0;
  do $r_1 < 2 \vee r_2 \leq 4;\ P_1\,!\,\mathit{acquire}() \rightarrow r_1 := r_1 + 1$
  [] $r_1 \leq 2 \vee r_2 < 4;\ P_2\,!\,\mathit{acquire}() \rightarrow r_2 := r_2 + 1$
  [] $P_1\,?\,\mathit{release}() \rightarrow r_1 := r_1 - 1$
  [] $P_2\,?\,\mathit{release}() \rightarrow r_2 := r_2 - 1$
  od;

The boolean guards ensure that the situation remains safe.

[It is assumed that the processes will not ask for more instances than their claimed maximum number.]