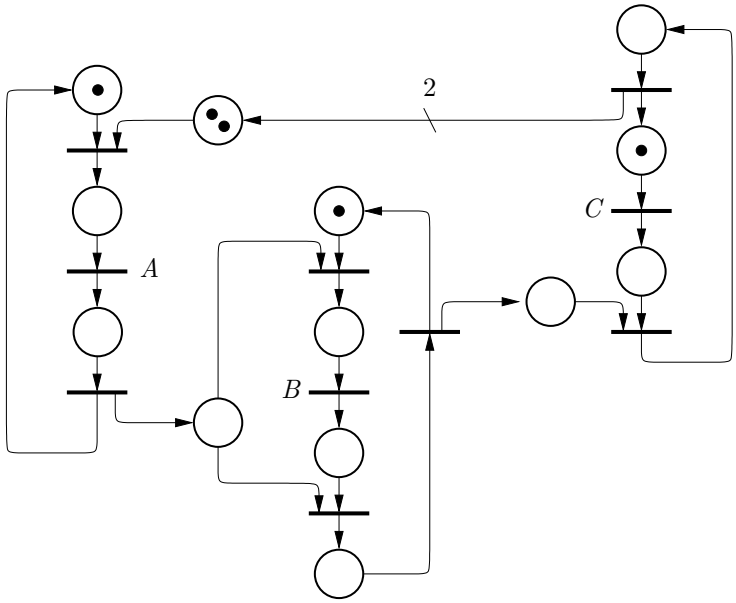
Suggested Solutions for

Written Exam, December 9, 2024

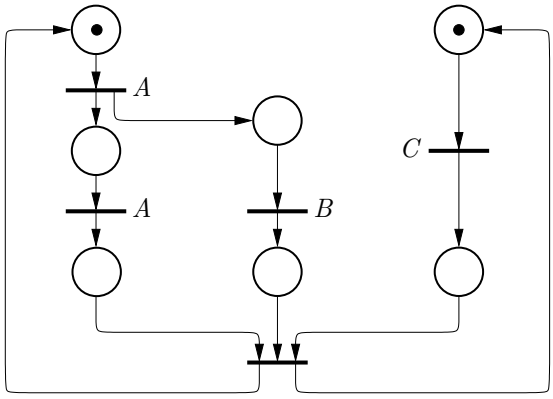
PROBLEM 1

Question 1.1

An almost[1] direct translation yields:



The synchronization may be succinctly expressed by:



Question 1.2

It is seen that B and C run concurrently, but synchronize in each round, i.e.

$$I \triangleq |b - c| \leq 1$$

[1]The two adjacent $\mathtt{V}(SA)$ operations have been coalesced.

Question 1.3

The above reduced Petri Net may be implemented using synchronous communications:

```
process P1;          process P2;          process P3;
  repeat               repeat               repeat
    A;                   P1 ? ();             C;
    P2 ! ();             B;                   P2 ? ();
    A;                   P3 ! ()              P1 ! ()
    P3 ? ()            forever              forever
  forever
```

PROBLEM 2

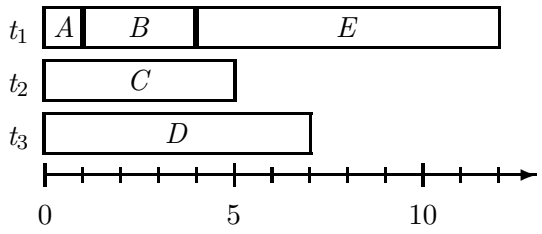
Question 2.1

(a) If the computation is split into $n = 5$ tasks, each task takes $T_1/5$ where T_1 is the sequential execution time. With $k = 3$ worker threads, these tasks are executed first three together, and then two together taking $2\times T_1/5$ in total. This gives a speedup of $T_1/(2\times T_1/5) = \mathbf{2.5}$.

(b) To achieve the best possible speedup, $\mathbf{k = 4}$ worker threads should be allocated and the number of tasks should be a (small) multiple of this, e.g. $\mathbf{n = 4}$. This will give a speedup of **4** which is the best achievable, using four processors.

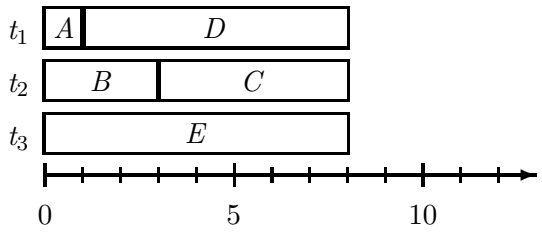
Question 2.2

(a) A worst-case scenario could be:



where the execution time becomes **12 seconds** resulting in a speedup of $\frac{24}{12} = \mathbf{2}$.

(b) Using three worker threads, 3 would be the best possible speedup. This may be achieved with the following scenario:



The resulting execution time is **8 seconds** yielding a speedup of $\frac{24}{8} = \mathbf{3}$.

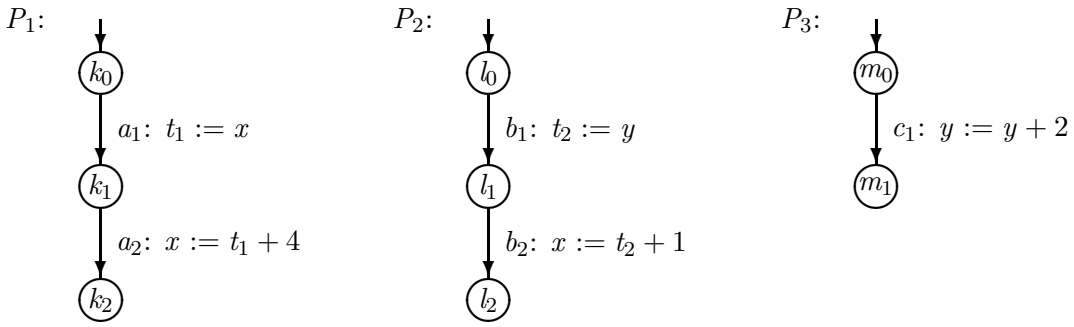
(c) [The above scenario may be generated by first submitting A, B, and E and then submitting D and C (just) when they should be started.]

$$submit(A);\ submit(B);\ submit(E);\ sleep(1);\ submit(D);\ sleep(2);\ submit(C)$$

PROBLEM 3

Question 3.1

(a) Transition diagrams:



[Location and action labels not required. Note that c_1 can be considered atomic.]

(b) [Rather than going through the 42 possible interleavings, we observe that the final value of x is determined by either a_2 or b_2. In the latter case, y may or may not have been set to 2 before being read, giving final values of 1 or 3. Otherwise, the final value set by a_2 depends on whether the value read for x in a_1 has been changed by b_2 or not. If not, x is set to 4. If it has been changed, it will have been read as 1 or 3 cf. the above, giving final values of 5 and 7.]

Analyzing the possible interleavings, it is found that the final value of x may be one of

$$1, 3, 4, 5, 7$$

Question 3.2

(a) P is preserved by a_3 only. [Not by a_1 if $x = -1$, and not by a_2 if $x = 0$.]

Q is preserved by a_2 and a_3. [Not by a_1 as it sets $x = 1$ while $y = 0$.]

R is preserved by a_1 and a_3. [Not by a_2 if x is negative. By a_3 since the guard ensures that x is positive and hence y will be so too.]

(b) I holds initially since $x = 0$.

Checking all atomic actions:

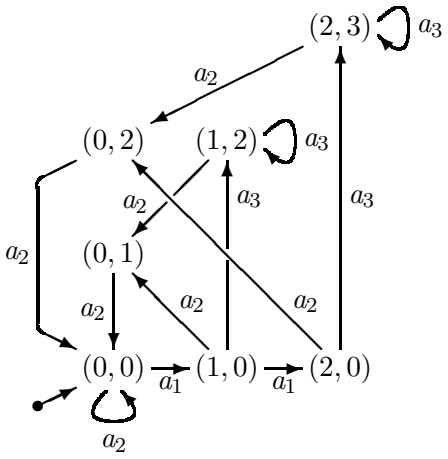
a_1: If executed, $x < 2$, so after the increment, $x <= 2$.

a_2: Setting x to 0 ensures that I holds.

a_3: Not potentially dangerous for I, as x is not changed.

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

(c) Transition graph:



(d) **Assuming weak fairness**

- F and H **do not** hold. The infinite execution

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

  satisfies weak fairness, but does not satisfy $x + y \geq 4$ nor $y = 2$ anywhere.
- G **does not hold**. Again the execution sequence $(*)$ satifies $\Box(x + y \neq 3)$, but never reaches $y = 3$.
- J **holds**. By inspecting the possible execution paths in the transition graph we observe that any weakly fair execution that meets $(2, 0)$ or $(2, 3)$ must eventually pass $(0, 2)$ with $y = 2$.

Assuming strong fairness

- F **does not** hold. The infinite execution

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

  satisfies strong fairness since all actions are executed. But $x + y$ never exeeceds 3.
- G **holds**. If $\Box(x + y \neq 3)$ it means that the state $(1, 2)$ is never visited any more. This rules out an exeuction like $(*)$ as this would enable a_3 infinitely often leading to $(1, 2)$. Therefore any execution for which $\Box(x + y \neq 3)$ must occationally pass $(2, 0)$ where a_3 is enabled. However, as a_3 is not taken to $(1, 2)$, it must be taken to $(2, 3)$ satisfying $y = 3$.
- H **holds**. As shown for G any execution will infinitely often enable a_3 leading either to $(1, 2)$ or to $(2, 3)$ followed by $(0, 2)$.
- J **holds**. Follows from weak fairness.

PROBLEM 4

Question 4.1

(a) The *Kit* component is readily implemented as a monitor:

```
monitor Kit
  var s : integer := 0;
      Pos : condition;
  procedure put(k : integer) {
    s := s + k;
    if s > 0 then signal(Pos);
  }
  function take() returns integer {
    var r : integer;
    while s ≤ 0 do wait(Pos);
    r := s;  s := 0;
    return r
  }
end
```

[Since at most one call of *take* can benefit from a *put*, a single *signal* suffices.]

(b) Calls of *take* should only wait if $s \leq 0$ unless some other calls have already been woken:

$$I \triangleq waiting(Pos) > 0 \Rightarrow s \leq 0 \vee woken(Pos) > 0$$

I holds initially as $waiting(Pos) = 0$.

In *put*: If s becomes positive while any waiting, one of these will be woken making the right hand side of I true.

In *take*: If a new call or a woken call waits it only does so if $s \leq 0$ ensuring I. If a call does not wait, but leaves the function, it sets $s = 0$ making I hold.

Since I holds initially and is preserved by both operations, I is an invariant of the monitor.

Question 4.2

```
process Control;
  var s, r : integer := 0;
  repeat
    in put(k : integer)    → s := s + k
    [] take() and s > 0 → r := s; s := 0; return r
    ni
  forever
```

Question 4.3

```
object Kit

  var s : integer := 0;
      e : semaphore := 1;
      pos : semaphore := 0;
      dpos : integer := 0

  procedure put(k : integer) {
    P(e);
    s := s + k;
    if s > 0 ∧ dpos > 0 then { dpos := dpos − 1;  V(pos) }
                        else  V(e)
  }

  function take() returns integer {
    var r : integer;
    P(e);
    if s ≤ 0 then { dpos := dpos + 1;  V(e);  P(pos) };
    r := s;  s := 0;
    V(e);
    return r
  }

end
```

Question 4.4

(a) Initially $Kit.put(-(n-1))$ is called.

Synchronization code for each process P_i $(i : 1..n)$:

$\vdots$
$Kit.put(1);$
$Kit.take();$
$Kit.put(1);$
$\vdots$

[In this solution initially a deficit of $n-1$ tokens is made. Each proces contributes one token upon arrival. When $n-1$ processes have arrived, the sum of the kitty has become 0. Therefore, when the last process arrives, the sum becomes 1 and one of the processes may pass the *take*() call. This process puts back one token to make the next process pass etc.]

Question 4,5

(a) Let *Kit* be initialized by the call $Kit.put(n)$.

Now, the readers and the writer may be synchronized by:

process $Reader[i : 1..n]$	**process** *Writer*
⋮	⋮
$Kit.put(Kit.take() - 1)$;	$Kit.put(-(n-1))$;
reading	$Kit.take()$;
$Kit.put(1)$;	writing
⋮	$Kit.put(n)$;
	⋮

[The value of s is used as a token count starting with n tokens. Each reader aquires one token by taking a positive number and returning all but one. After reading, the token is put back. The writer first claims $n-1$ tokens and then waits for the last one itself. Once finished, the writer returns all tokens. Alternatively, before writing, the writer may repeatedly call *take* till all n tokens have been obtained.]

(b) Under the standard assumption of *weak fairness*, it is seen that the value s of *Kit* will repeatedly become positive if used as in (a). However, if only weak fairness is assumed for the *take* operation, neither the readers nor the writer are guaranteed to pass the call of *take* and hence the solution **is not fair** to any of them.

If *strong fairness* is assumed for the *take* operation, a call of *take* will eventually finish (as the guard will be infinitely often true) and hence the solution **will be fair** to both the readers and the writer.

PROBLEM 5

Question 5.1

```
monitor Broadcast

  var message : Msg;
      rem : integer := 0;              // Receivers remaining in ongoing broadcast
      Senders : condition;
      Receivers : condition;

  procedure send(m : Msg) {
    while empty(Receivers) ∨ rem > 0 do wait(Senders);
    rem := min(length(Receivers), Lim);
    message := m;
    signal(Receivers)
  }

  function listen() returns Msg {
    if rem = 0 then signal(Senders);
    wait(Receivers);
    rem := rem − 1;
    if rem > 0 then signal(Receivers)
               else if ¬empty(Receivers) then signal(Senders);
    return message
  }

end
```

Question 5.2

The new module is called *Multicast*:

```
module Multicast
  op send(m : Msg);
  op listen() returns Msg;
  op set(v : posinteger);
body

  process Control;
    var K : posinteger := Lim;
    repeat
      in send(m : Msg) and ?listen ≥ K → for i in 1..K do
                                           in listen() → return m ni
      []  set(v : posinteger)                  → K := v
      ni
    forever

end Multicast;
```