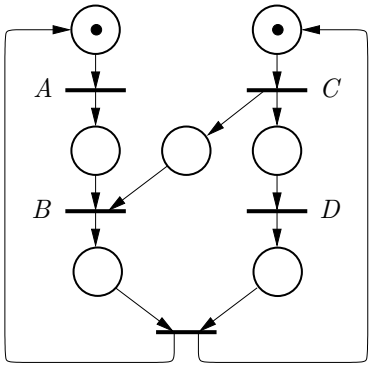
Suggested Solutions for

Written Exam, December 9, 2020

PROBLEM 1

Question 1.1

A direct translation is readily made. From this, the following recduced Petri net may be derived:



Question 1.2

From the Petri Net it is seen that A and D may be executed concurrently, but are synchronized in each round. Hence:

$$I \triangleq |a - d| \leq 1$$

Question 1.3

The above Petri Net may be implemented as follows:

process P_1;	**process** P_2;	**process** P_3;	**process** P_4;
repeat	**repeat**	**repeat**	**repeat**
A;	$P_3\,?\,()$;	C;	$P_3\,?\,()$;
$P_2\,!\,()$;	$P_1\,?\,()$;	$P_2\,!\,()$;	D;
$P_2\,?\,()$	B;	$P_4\,!\,()$;	$P_2\,?\,()$;
forever	$P_4\,!\,()$;	$P_4\,?\,()$	$P_3\,!\,()$
	$P_1\,!\,()$	**forever**	**forever**
	forever		

[Special care must be taken to let D execute after C without waiting for A. Here, this is achieved by letting P_2 synchronize with P_3 before synchronizing with P_1, but other schemes are also possible.]

PROBLEM 2

Question 2.1

(a) The statement pairs are checked for critical references with respect to each other:

Pair	Mutually atomic	Rationale
a, b	NO	Two critical references in both a and b.
a, c	NO	Both reading and writing of x in a are critical.
a, d	YES	Writing to y is the only critical reference in d.
b, c	YES	Only one critical reference in both b and c
b, d	NO	Two critical references to y in d.
c, d	YES	No critical references — totally independent.

(b) Going through the six possible interleavings of the two atomic actions, the possible final values of (x, y) are found to be:

$$(3, 2), (3, 5), (5, 2)$$

Question 2.2

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

All three a-actions are potentially dangerous for I:

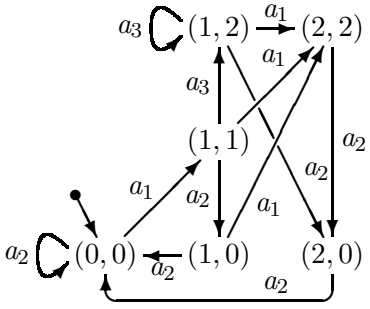
a_1: After the execution, $x = y$. Therefore, if $x = 0$ also $y = 0$ and I holds.

a_2: After this action, $y = 0$, i.e. I will hold.

a_3: This action is executed only if $x = 1$ and since x is not changed by the action, after the execution x is still non-zero and hence I holds.

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

(b) Transition diagram:



(c) **Assuming weak fairness**

- F **does not** hold. The only reachable state satisfying $y > x$ is $(1, 2)$. This may be avoided by the following execution sequence:

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

  Since this execution path does not remain at a single state, it satisfies weak fairness.
- G **does not** hold. The cycle $(*)$ also avoids $x = 2$.
- H **does** hold. Any execution will repeatedly pass through $(1, 0)$ or $(2, 0)$.

- J **does not** hold. The only state satisfying $x + y = 4$ is $(2, 2)$. The cycle $(*)$ passes through $(1, 1)$. but avoids $(2, 2)$.

Assuming strong fairness

- F **holds**. As $(1, 1)$ is reached infinitely often, a_3 is enabled infinitely often and hence $(1, 2)$ is reached infinitely often.
- G **holds**. As $(1, 2)$ is reached infinitely often (cf. F), also $(2, 0)$ must be reached infinitely often satisfying $x = 2$.
- H **holds**. By weak fairness.
- J **does not** hold. The execution cycle

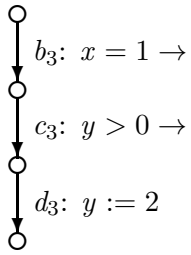
$$(0,0) \xrightarrow{a_1} (1,1) \xrightarrow{a_3} (1,2) \xrightarrow{a_2} (2,0) \xrightarrow{a_2} (0,0) \xrightarrow{a_1} \cdots \qquad (**)$$

satisfies strong fairness, and passes through $(1, 1)$ but still escapes $(2, 2)$.

(d) If a_3 cannot be considered atomic as a whole, by the default assumption of atomic reads and writes and assuming that the guard is evaluated from left to right, it will correspond to

$$b_3\colon \langle\, \textbf{await } x = 1 \,\rangle; \quad c_3\colon \langle\, \textbf{await } y > 0 \,\rangle; \quad d_3\colon \langle\, y := 2 \,\rangle$$

This can be depicted by the transition diagram:



(e) Given the refinement above, the interleaving

$$(0,0) \xrightarrow{a_1} (1,1) \xrightarrow{b_3} (1,1) \xrightarrow{c_3} (1,1) \xrightarrow{a_2} (1,0) \xrightarrow{a_2} (0,0) \xrightarrow{d_3} (0,2)$$

violates I.

PROBLEM 3

Question 3.1

The *WG* component is readily implemented as a monitor using cascade wakeup:

```
monitor WG

  var count : integer := 0;
      nonPos : condition;

  procedure addWait(k : integer) {
    while count > 0 do wait(nonPos);
    count := count + k;
    if count ≤ 0 then signal(nonPos);
  }

  procedure add(k : integer) {
    count := count + k;
    if count ≤ 0 then signal(nonPos);
  }

end
```

Question 3.2

```
object WG

  var count : integer := 0;
      nd : integer := 0
      e : Semaphore := 1;
      d : Semaphore := 0;

  procedure addWait(k : integer) {
    P(e);
    if count > 0 then { nd := nd + 1;  V(e);  P(d) }
    count := count + k;
    if count ≤ 0 ∧ nd > 0 then nd := nd − 1;
                               V(d)
                          else V(e)
  }

  procedure add(k : integer) {
    P(e);
    count := count + k;
    if count ≤ 0 ∧ nd > 0 then nd := nd − 1;
                               V(d)
                          else V(e)
  }

end
```

Question 3.3

process *Server*

var *count* : *integer* := 0;

do $count \leq 0$; $Client[*]\,?\,AddWait(k) \rightarrow count := count + k$
[] $Client[*]\,?\,Add(k) \rightarrow count := count + k$
od

Question 3.4

The main program may look like:

$WG.add(3)$;
start $\{SL_1;\ \ WG.add(-1)\ \}$;
start $\{SL_2;\ \ WG.add(-1)\ \}$;
start $\{SL_3;\ \ WG.add(-1)\ \}$;
$WG.waitAdd(0)$;

Question 3.5

Synchronization code for each process:

$\vdots$
$WG.waitAdd(1)$;
critical section
$WG.add(-1)$
$\vdots$

[In general WG may be considered a general semaphore initialized to 1 with $waitAdd(1)$ and $add(-1)$ acting as `P` and `V` operations respectively.]

Question 3.6

Let M be an integer constant such that $M \geq (N-1)$ and let WG be initialized by the call $WG.add(-M)$.

Now, the readers and writers may be synchronized by:

process *Reader*$[i : 1..N]$

$\vdots$
$WG.waitAdd(1)$;
reading
$WG.add(-1)$
$\vdots$

process *Writer*

$\vdots$
$WG.add(M)$;
$WG.waitAdd(1)$;
writing
$WG.add(-(M+1))$;
$\vdots$

Initalizing *count* to $-M$ ensures that all N readers are allowed to run concurrently. The call $WG.add(M)$ prevents new readers from starting and when all readers are gone, $count = 0$ and $WG.wait(1)$ may be passed, blocking for readers. The writer may still suffer from starvation though.

PROBLEM 4

Question 4.1

(a) Calls of *get* should wait only if the most recent problem has been solved. As the latter is represented by the *done* variable, this can be expressed by:

$$I \triangleq \mathit{waiting}(\mathit{newProblem}) > 0 \Rightarrow \mathit{done}$$

(b) Initially I holds as *newProblem* is empty.

In *solve*: Before entering the wait, *done* is set to false, but as *newProblem* is emptied before the wait, I will hold. After the wait, the invariant is not changed whether a new wait is made or the function left.

In *get*: If the *newProblem* queue is entered this happens only if *done* is true, satisfying I. Otherwise the function is left with I unchanged.

In *result*: Either *done* is unchanged or set to true. In any case, I is preserved.

As I holds initially and I is preserved by all stretches of activity, I is a monitor invariant.

Question 4.2

To implement the functionality of $\mathit{solve}(x, k)$, the number of problem copies fetched by *get* must be constrained. For this, the following changes suffice:

- The variable $\mathit{remaining} : \mathit{integer} := 0$ is added to the variable declarations.
- The parameter k is added to *solve*: $\mathit{solve}(x : X, k : \mathit{posinteger})$
- In *solve*, a statement $\mathit{remaining} := k;$ is inserted anywhere before the while loop.
- The implementation of *get* is changed to:

```
function get() returns (X, integer) {
    while done ∨ remaining = 0 do wait(newProblem);
    remaining := remaining − 1;
    return (prob, no)
}
```

Question 4.3

(a) Suppose one user process calls $\mathit{solve}(x_1)$, some worker process gets $(x_1, 1)$ and starts working on it. Now another user process calls $\mathit{solve}(x_2)$, overwriting *prob*. A second worker process may then obtain $(x_2, 2)$ in *get*, quickly find a solution y_2 and call $\mathit{result}(y_2, 2)$. This will wake up the first user process which picks up solution y_2 and returns it for problem x_1.

(b) The general technique of using a pre-queue can be applied. First, the pre-queue and a flag must be declared:

```
var solving : boolean := false;
    notInUse : condition;
```

Using these, the whole body of *solve* (except the final **return**) may be wrapped into an outer critical section:

```
procedure solve(x : X) returns Y
  while solving do wait(notInUse);
  solving := true;
    ⋮
  solving := false;
  signal(notInUse);
  return sol
```

Question 4.4

In order to stop working on a problem already solved, the worker processes may poll *ParSolve* in each algorithm iteration using the current version number and skip the rest of the algorithm (as well as the result delivery) if the problem is reported as already solved.

The poll operation could be added as

```
procedure isSolved(ver : integer) returns boolean {
  return ver ≠ no ∨ done;
}
```

Question 4.5

(a)

```
process Control;
  var no : integer := 0;
      done : boolean;
      sol : Y;
  repeat
    in solve(x : X) returns Y →
          no := no + 1;
          done := false;
          while ¬done do
            in get() returns (X, integer)    → return (x, no)
            [] result(y : Y, ver : integer)  → if ver = no then { done := true;
                                                                  sol := y      }
            ni;
          return sol;
    [] result(y : Y, ver : integer) → skip
    ni
  forever;
```

(b) As only one call of *solve* is accepted at a time, the module would work correctly also if *solve* was called by several concurrent user processes.