Suggested Solutions for

Written Exam, December 6, 2023

PROBLEM 1

Question 1.1

(a) The speedup cannot exceed the number of processors nor worker threads, hence **5** is the upper limit determined by the number of threads.

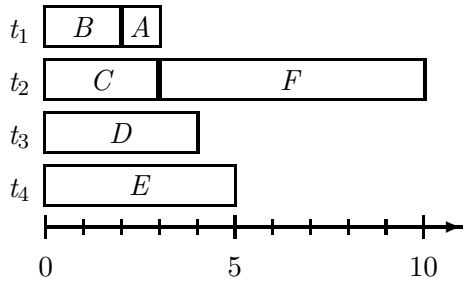
(b) If the computation is split into only $t = 2^2 = 4$ tasks, allocating four threads may yield a speedup of 4. To enable a greater speedup, the computation should be split into more tasks, e.g. $t = 2^3 = \mathbf{8\ tasks}$, each taking $T_1/8$ time, where T_1 the serial execution time.

[If six worker threads were allocated for the thread pool, they will first work on six of the tasks for $T_1/8$ time units and then two of the threads would work on the remaming two tasks, again for for $T_1/8$ time units. This would only give a speedup of $T_1/(2*(T_1/8)) = \mathbf{4}$.]

If we assume that the computation takes much longer time than the period over which scheduling becomes evenly distributed, allocating **eight worker threads** will make each thread seem to work at a reduced execution rate. All eigth tasks may now be carried out at the same time, but at the reduced rate of 6/8. Hence the computation time becomes $T_1/8 * (6/8)^{-1}$ corresponding to an expected **speedup of 6**. This is the best obtainable on this system.

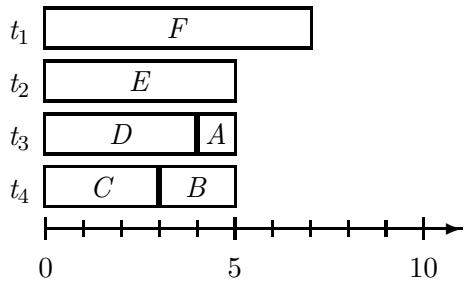
Question 1.2

(a) The worst scheduling scenario occurs when the long task F is executed as the last one:



Here, the execution time becomes 10 seconds corresponding to a speedup of $22/10 = \mathbf{2.2}$.

(b) The best possible execution time is 7 seconds [due to F] which could occur in a scenario like:

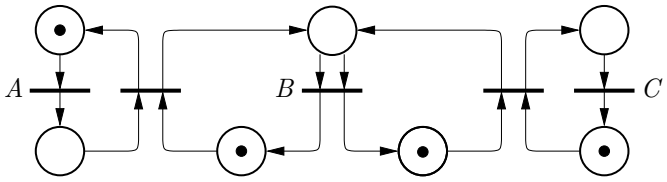


where the tasks could have been submitted in reverse alphabetical order: F, E, D, C, B, and A. This will give a speedup of $22/7 \approx 3,14$ [$\approx \pi(!)$] > 3.

PROBLEM 2

Question 2.1

[A direct translation of the semaphore-based program to a Petri Net is readily made. This may be reduced to:]



Question 1.2

Apart from the first A, both A and C must each synchronize with B. This implies $|(a-1)-c| \leq 2$ equivalent to:

$$I \triangleq c-1 \leq a \leq c+3$$

Question 2.3

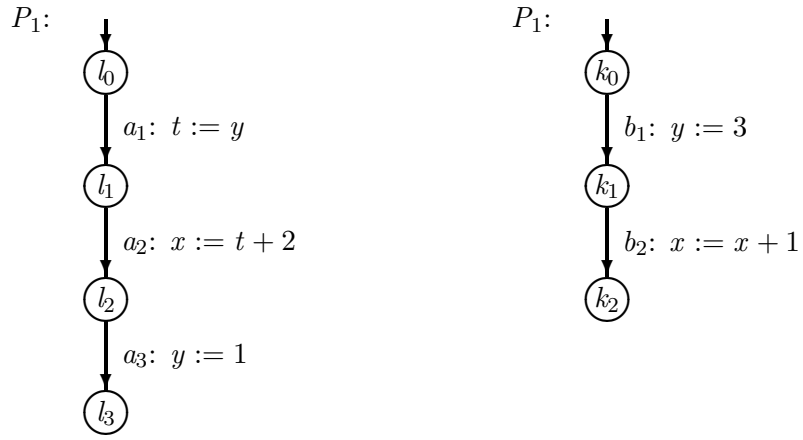
[Before B, synchronization with both A and C must be carried out:]

```
process P1;          process P2;                    process P3;
  repeat               repeat                         repeat
    A;                   if P1?() → P3?()               P2!() :
    P2!()                [] P3?() → P1?()               C
  forever                fi;                          forever
                         B
                       forever
```

PROBLEM 3

Question 3.1

(a) Transition diagrams:



[Location and action labels not required.]

(b) The final values of (x, y) can be found by going through the 10 possible interleavings. [We may observe that y is determined by either a_3 or b_1. In the latter case all of P_1 comes before P_2 resulting in $(x, y) = (3, 3)$ Otherwise P_1 may either read y before or after it has been set to 3 and the incrementation of x may be overwritten or not. This leads to the following combinations:]

$$(3,3), (2,1), (3,1), (5,1), (6,1)$$

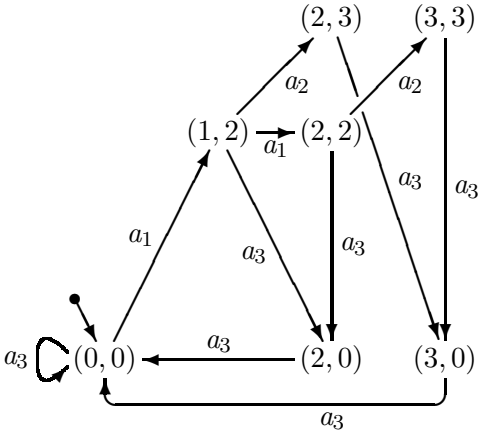
Question 3.2

(a) P is preserved by a_2 and a_3. By a_2 since it increments both x and y by 1 if executed. By a_3 since x becomes y and y becomes 0. Not by a_1 since it may take $(0, 0)$ to $(1, 2)$ not satifying P.

Q is preserved by a_2, but not by a_1 (same as case as above) and not by a_3 e.g. for $(x, y) = (1, 1)$.

R is preserved by a_1 as it cannot decrease y when $x \leq 1$. R is also preserved by a_2, incrementing x and y, but not by a_3 e.g. for $(x, y) = (1, 0)$.

(b) Transition graph:



(c) From the transition graph if is seen that the only reachable states in which $y \geq 2$ are $(1,2),(2,2),(2,3)$, and $(3,3)$ all satisfying $1 \leq x \leq y$. Hence I is an invariant of the program.

(d) **Assuming weak fairness**

- F **holds**. From any reachable state, the execution must eventually reach $(0,0)$, then $(1,2)$ and from there either $(2,0)$, $(2,2)$, or $(2,3)$ all satifying $x = 2$.
- G **does not** hold. From any of the states satifying $x = 2$ we may take a_3 to $(0,0)$ and from there avoid $x = 3$ e.g. by the cycle

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

- H **does not** hold. The only reachable states satisfying $x + y = 4$ and $x + y = 6$ are $(2,2)$ and $(3,3)$ respectively. However from $(2,2)$ the execution may again follow the cycle $(*)$, never reaching (3,3).
- J **does not** hold. The execution may follow the cycle $(*)$ avoiding $y = 3$ for ever.

Assuming strong fairness

- F **holds**. By weak fairness.
- G **holds**. In any execution the state $(1,2)$ must be passed over and over again and hence the action a_2 must be enabled infinitely often. Due to strong fairness, a_2 must be eventually executed (from some state) leading either directly to $(3,3)$ or to $(2,3)$ from which $(3,0)$ is reached.
- H **does not** hold. From state (2,2) the execution may first move to (0,0) by a_3 and from there follow the cycle

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

  which satisfies strong fairness for all actions, but avoids $(3,3)$ for ever.
- J **holds**. As for G, from any point in any execution a_2 must be executed eventually leading to $y = 3$.

PROBLEM 4

Question 4.1

var s : *integer* := 0;

pass(k : *posinteger*) : $\langle\, s \geq k \rightarrow s := s - k \,\rangle$;

release(k : *posinteger*) : $\langle\, s := s + k \,\rangle$;

Question 4.2

(a) Initially: I holds as $s = 0$.

Consider the following stretches of activity:

In *pass*: If, upon entering, $s < k$, the call waits, nothing has changed and I is preserved. The same applies if the call waits again after being awakened. If the call passes the **while** directly or after a wait, s must be at least k and hence decrementing it by k will no invalidate I.

In *release*: As k is positive, s will be incremented, preserving I.

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

(b) No waiting call of *pass* must have request that could be satisfied:

$$J \triangleq \mathit{waiting}(\mathit{pass}) > 0 \Rightarrow s < \min(\mathit{params}_{\mathit{pass}}(\mathit{queue}))$$

(c) If only *signal* was used, the process awakened could have a request that would exceed the incremented s while other processes might have lower, satifiable requests. Also, there could be more than one call of *pass* that could be satisfied after incrementing s.

(d) If calls of *pass*(k) wait with rank k, the signalling may be reduced, e.g. by doing a cascade wakeup as long as the smallest request can be satisfied:

monitor *GenSem*

var s : *integer* := 0;
queue : *condition*;

procedure *pass*(k : *posinteger*) {
while $s < k$ **do** *wait*(*queue*, k);
$s := s - k$
if $\neg$*empty*(*queue*) $\wedge$ $s \geq$ *minrank*(*queue*) **then** *signal*(*queue*)
}

procedure *release*(k : *posinteger*) {
$s := s + k$;
if $\neg$*empty*(*queue*) $\wedge$ $s \geq$ *minrank*(*queue*) **then** *signal*(*queue*)
}

end

[Making the cascade by just calling *signal*(*queue*) at the end of *pass* and *release* would also reduce the signalling, but then a single unnecessary wakeup might occur at the cascade ending.]

Question 4.3

```
process Control;
  var s : integer := 0;
  repeat
    in pass(k : posinteger) and s ≥ k → s := s − k
    [] release(k : posinteger       → s := s + k
    ni
  forever;
```

Question 4.4

(a) Initially, *GenSem.release*(n) should be called.

Readers:		Writers:	
	GenSem.pass(1);		*GenSem.pass*(n);
	reading		writing
	GeSem.pass(1);		*GenSem.release*(n);

(b) As there may be active readers at all times such that $s < n$, writers may be starved.

On the other hand, a sequence of writers may prevent any reader from starting as the order in which processes reenters the monitor after being woken with *signal_all* is not defined.

Therefore, the solution is neither fair towards readers nor writers.

[If the FCFS version of Question 4.6 below was used, the solution would be fair to both readers and writers.]

Question 4.5

(a) No initialization code is necessary.

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

```
⋮
GenSem.release(1);
GenSem.pass(n);
GenSem.release(n);
⋮
```

[In this solution each process contributes one token. When n tokens are available the processes may each discover this by grabbing n tokens and putting them back again.]

(b) In order to use the *GenSem* for another synchronization point, the value of s must be reset to 0. This cannot done safely until all processes are known to have **passed** the barrier. Therefore the solution cannot be used for repeated synchronization.

[A cyclic barrier may be made using *three* instances of *GenSem* and reset the various instances at the right stages.]

Question 4.6

[In order to know the demand of the first call of *pass* a separate condition queue *front* is used for that, setting its request in the variable *req*. Any following calls wait in condition queue *back*. The value of *req* is also used to indicate whether there is a call of *pass* waiting in *front* ($req > 0$) or whether a call of *pass* has been woken from *back* and is on its way to take over the front ($req < 0$). If $req = 0$ there are no waiting calls of *pass*.]

monitor *FCFS_GenSem*

var s : *integer* := 0;
req : *integer* := 0;
front, *back* : *condition*;

procedure *pass*(k : *posinteger*) {
if $req \neq 0$ **then** *wait*(*back*);
$req := k$;
if $s < k$ **then** *wait*(*front*);
$s := s - k$
if $\neg$*empty*(*back*) **then** {$req := -1$; *signal*(*back*) }
else {$req := 0$ }
}

procedure *release*(k : *posinteger*) {
$s := s + k$;
if $req > 0 \wedge s \geq req$ **then** *signal*(*front*)
}

end

Aside

The `java.util.concurrent.Semaphore` implements such a generalized semaphore. If the *fairness* attribute is set at creation time, the semaphore becomes FCFS.

PROBLEM 5

[While handling a call of *decide* the server must hand out the problem to any calls of *get* for which the problem is new. Meanwhile any results for this problem must be recorded until m results have been obtained. Here the approvals/non-approvals are counted in two variables *infavour* and *against*. While no decision is being made, obsolete results must still be discarded.]

```
process Control;
  var no : integer := 0;
      infavour, against : integer;
  repeat
    in decide(d : D) returns boolean →
          no := no + 1;
          infavour := 0;  against := 0;
          while infavour + against < m do
            in get(prev : integer) returns (D, integer)
                                        and no > prev → return (d, no)
            [] result(ok : boolean, ver : integer)          →
                                    if ver = no then
                                      if ok then infavour := infavour + 1;
                                           else against  := against + 1
            ni;

          return infavour ≥ 2 * against

    [] result(ok : boolean, ver : integer) → skip
    ni
  forever;
```