Suggested Solutions for

Written Exam, December 7, 2022

PROBLEM 1

Question 1.1

(a) With the serial fraction $f = 10\,\% = 0.1$ and the number of processors $p = 6$, the speedup follows from substitution into Amdahl's Law:

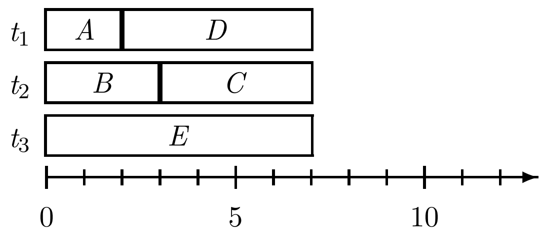
$$S = \frac{1}{f + \frac{(1-f)}{p}} = \frac{1}{0.1 + \frac{0.9}{6}} = \frac{1}{0.25} = \mathbf{4}$$

(b) Likewise, the overall limit for the speedup is given by $S_{\mathsf{max}} = \frac{1}{f} = \frac{1}{0.1} = \mathbf{10}$

Question 1.2

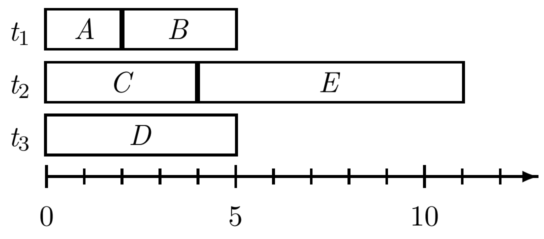
(a) The speedup cannot exceed the number of processors nor worker threads, hence **3** is the upper limit determined by the number of threads.

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



The resulting execution time is **7 seconds** yielding a speedup of $\frac{21}{7} = \mathbf{3}$.

(c) A worst-case scenario could be:

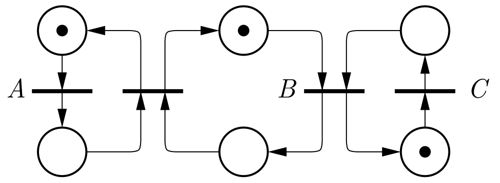


where the execution time becomes **11 seconds** resulting in a speedup of $\mathbf{\frac{21}{11}}$ [≈ 1.91].

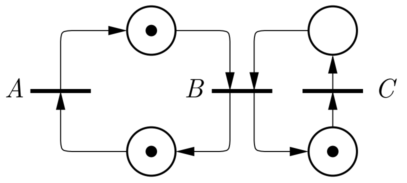
PROBLEM 2

Question 2.1

[From the inequations, it is recognized that A and B must be executed in lock-steps whereas B and C must alternate, starting with C. These two synchronizations may be combined as in:]



[Alternatively the lock-step execution of A and B may be represented by:



Here, A may fire together with itself, but the invariant is still satisfied.]

Question 2.2

[The locks-step synchronization may be implemented by cross-signalling:]

```
var $S_A, S_{BA}, S_{BC}, S_C$ : semaphore;
$S_A$ := 0;  $S_{BA}$ := 0;  $S_{BC}$ := 0;  $S_C$ := 0;

process $P_A$;        process $P_B$;        process $P_C$;
  repeat                repeat                repeat
    $A$;                    P($S_{BC}$);          $C$;
    V($S_{BA}$);            $B$;                  V($S_{BC}$);
    P($S_A$)                V($S_C$);             P($S_C$)
  forever                   V($S_A$);         forever
                            P($S_{BA}$)
                        forever
```

Question 2.3

[Here the lock-step is readily implemented with synchronous communication:]

```
process $P_1$;        process $P_2$;        process $P_3$;
  repeat                repeat                repeat
    $A$;                    $P_3$ ? ();           $C$;
    $P_2$ ! ()              $B$;                  $P_2$ ! ();
  forever                   $P_3$ ! ();           $P_2$ ? ()
                            $P_1$ ? ()        forever
                        forever
```

PROBLEM 3

Question 3.1

(a) Transition diagrams:

P_1:

$$\begin{tikzcd} {} \arrow[d] \\ k_0 \arrow[d, "a_1:\ x := y+1"] \\ k_1 \end{tikzcd}$$

P_2:

$$\begin{tikzcd} {} \arrow[d] \\ l_0 \arrow[d, "b_1:\ t := x+y"] \\ l_1 \arrow[d, "b_2:\ y := t+2"] \\ l_2 \end{tikzcd}$$

P_3:

$$\begin{tikzcd} {} \arrow[d] \\ m_0 \arrow[d, "c_1:\ x := 4"] \\ m_1 \end{tikzcd}$$

[Location and action labels not required. Note that b_1 can be considered atomic.]

(b) [Rather than going through the 12 possible interleavings, we observe that the final value of x is determined by either a_1 or c_1. In the latter case, x becomes 4. Otherwise, the final value set by a_1 is based on the value of y. When read, y may or may not have been changed by b_2. If changed by b_2 the updated value of y depends on whether x was changed by c_1 before or after b_1. This gives us three different combinations to investigate, leading to the results 1, 3, and 7.]

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

$$1, 3, 4, 7$$

Question 3.2

(a) P is preserved by a_1 and a_2. By a_1 since it increments y. Similarly for a_2 with x. Not by a_3 if y is negative.

Q is preserved by a_1 and a_2. By a_1 since it increments y. By a_2 since the guard $x \neq y$ ensures $y > x$ before the execution and hence $y \geq x$ after increment of x. Not by a_3 if y is positive.

R is preserved by a_1 only. The guard prevents a_1 from executing. For a_2, x may be equal to $y - 1$ before the action making them equal after execution. If $y = 0$, both x and y will be 0 after the execution of a_3.

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

Checking all atomic actions:

a_1: Before the execution, $x = y$ and $x < 2$, hence $y < 2$ or equivalently $y \leq 1$. Incrementing y by one therefore cannot violate neither $y \leq 2$ nor $0 \leq y$ and hence I holds.

a_2: Not a potentially dangerous for I, since y is not changed.

a_3: Always after this action, $y = 0$, i.e. I holds.

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

(c) Transition graph:

$$\begin{tikzcd}
 & (1,2) \arrow[r, "a_2"] \arrow[rdd, "a_3"] & (2,2) \arrow[dd, "a_3"] \\
(0,1) \arrow[r, "a_2"] \arrow[rd, "a_3"] & (1,1) \arrow[u, "a_1"] \arrow[d, "a_3"] & \\
(0,0) \arrow[u, "a_1"] \arrow[loop left, "a_3"] & (1,0) \arrow[l, "a_3"] \arrow[r, "a_2"] & (2,0) \arrow[ll, bend left, "a_3"]
\end{tikzcd}$$

(d) **Assuming weak fairness**

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

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

  satisfies weak fairness, but does not satisfy $x + y \geq 2$ nor $x = 2 \vee y = 2$.

- H **holds**. By inspecting the possible execution paths in the transition graph we observe that any execution must repeatedly get back to $(x, y) = (0,0)$ and from there by a_1 to $(0,1)$ and then either by a_2 to $(1,1)$ or by a_3 to $(1,0)$.

- G **holds**. Among the reachable states $y = 2$ holds only in $(1,2)$ and $(2,2)$. From $(1,2)$ either a_2 leads to $(2,2)$ or a_3 leads to $(2,0)$. Thus, whenever $y = 2$, x will be or become two.

Assuming strong fairness

- G and H **hold**. Follows from weak fairness.

- F **holds**. The only way to avoid the states for which $x + y \geq 2$ would be to remain in the infinite loop

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

  For this execution, however, a_2 is enabled infinitely often and therefore must be taken leading to either $(1,1)$ or $(2,0)$.

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

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

  satisfies strong fairness since all actions are executed. But neither $x = 2$ nor $y = 2$ occurs.

PROBLEM 4

Question 4.1

(a) When the protocol schemes are followed, for each of the variables r, u, and w, the variable is alternatingly incremented and decremented (starting with incrementation). Since the variables are initially 0, the invariant H follows.

(b)

$$I \triangleq u + w \leq 1 \land (r = 0 \lor w = 0)$$

(c) If multiple update locks were allowed to be taken, the corresponding *upgradeU* operations would have to wait for the other update locks to be released, leading to deadlock.

Question 4.2

(a)

```
monitor Lock

  var $r, u, w$ : integer := 0;
      $OkRd, OkUp, OkWr$ : condition;

  procedure lockR() {
    while $w > 0$ do wait(OkRd);
    $r := r + 1$
  }

  procedure unlockR() {
    $r := r - 1$;
    if $r = 0$ then signal(OkWr)
  }

  procedure lockU() {
    while $u + w > 0$ do wait(OkUp);
    $u := u + 1$
  }

  procedure upgradeU() {
    while $r > 0$ do wait(OkWr);
    $u := u - 1$;  $w := w + 1$
  }

  procedure unlockU() {
    $w := w - 1$;
    signalAll(OkRd);
    signal(OkUp)
  }

end
```

[When writing is done, all waiting readings may be started together with a single update.]

(b) [A call of *upgradeU*() should wait on *OkWr* only when there are still readings active:]

$$J \triangleq \mathit{waiting}(\mathit{OkWr}) > 0 \Rightarrow r > 0$$

J holds initially, because *OkWr* is empty. In *upgradeU*, *OkWr* is entered only when $r > 0$. Whenever r becomes 0, the *OkWr* queue is signalled and hence will be emptied, as *upgradeU* can be called by at most one update operation.

(c)
```
procedure $dropU()$ {
  $u := u - 1$;
  $signal(OkUp)$
}
```

Question 4.3

(a) The call of *lockU*() ensures there there are no other potential writings taking place and the call of *upgradeU* ensures that there are no readings active. Hence the effect of taking a write lock is achieved.

(b) Initialization code: **for** i **in** $1..n$ **do** *L.lockR*();

Synchronization code for each process P_i $[i : 1..n]$:

```
⋮
$L.unlockR()$;
$L.lockU()$;
$L.upgradeU()$;
$L.unlockU()$;
⋮
```

[The last three operations may be seen as taking a write lock and releasing it again. This may be passed, when all the initial read locks have been released by arrival at the barrier.]

(c) Once all the initial read locks have been released, the write lock sequence may be passed an arbitrary number of times and hence the above code **cannot** be used for repeated synchronization. Also, it would not follow the protocol scheme for readings by calling *unlockR*() several times without *lockR*() in between.

[To use the code for repeated synchronization, the read locks would have to be taken again, but only after all processes have passed the barrier. This may be done using *three stages* each stage being an instance of the above code.]

Question 4.4

(a) The specification can be implemented directly by a server process:

```
process $Control$;
  var $r, u, w : integer := 0$;
  repeat
    in $lockR()$ and $w = 0$              $\rightarrow r := r + 1$
    [] $unlockR()$                        $\rightarrow r := r - 1$
    [] $lockU()$ and $u + w = 0$          $\rightarrow u := u + 1$
    [] $upgradeU()$ and $r = 0$           $\rightarrow u := u - 1;\ w := w + 1$
    [] $unlockU()$                        $\rightarrow w := w - 1$
    ni
  forever
```

(b) The server can be made fair to updates by blocking for readings when an update lock can be taken or when an upgrade is pending. To be fair to readings, these may be processed after each successful update.

```
process $FairControl$;
  var $r, u, w : integer := 0$;
  repeat
    while $w = 0$ do
      in $lockR()$ and $(?lockU = 0 \lor u > 0) \land\ ?upgradeU = 0 \rightarrow r := r + 1$
      [] $unlockR()$                  $\rightarrow r := r - 1$
      [] $lockU()$ and $u = 0$        $\rightarrow u := u + 1$
      [] $upgradeU()$ and $r = 0$     $\rightarrow u := u - 1;\ w := w + 1$
      ni;
    in $unlockU() \rightarrow w := w - 1$ ni;
    while $?lockR > 0$ do  in $lockR() \rightarrow r := r + 1$ ni
  forever
```

[Here, the writing phase is handled separately in order to readily clear the readings queue.]

Question 4.5

```
$lockR()$ :       P($SR$)
$unlockR()$ :     V($SR$)
$lockU()$ :       P($SU$)
$upgradeU()$ :    for $i$ in $1..m$ do P($SR$)
$unlockU()$ :     for $i$ in $1..m$ do V($SR$)
                  V($SU$)
```

[The solution is based on a standard semaphore solution for readers and writers, except that the wait for cease of reading is deferred till the upgrade step.]

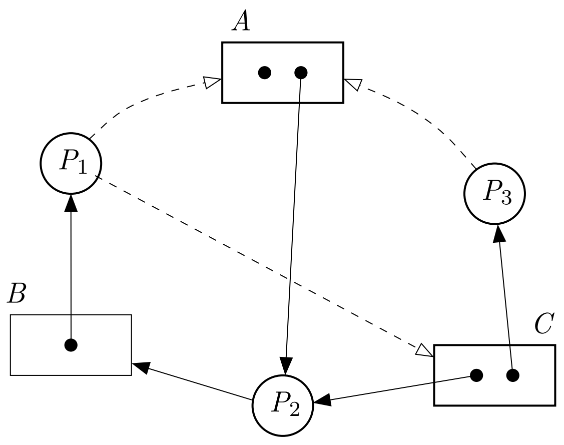
Aside: The Boost C^{++} library provides an upgradable lock like L.

PROBLEM 5

Question 5.1

(a) Channel contents: C_A : A^2 C_B : empty C_C : empty

(b) Resource allocation graph:



(c) The remaining A instance may be granted to P_3 which may then finish releasing a C instance. Then P_1 may finish and release the B instance and finally P_2 may finish. Since all processes can finish, the situation is considered *safe*.

(d) If the remaining A instance is requested by and granted to P_1, then when P_3 subsequently requests an A resource, all processes have pending requests while there are no free resource instances. Hence the situation is in *deadlock*.

(e) The following neighbour acquisitions exchanges may be made:

In P_1, **receive** $C_A(a)$ and **receive** $C_C(c)$ are exchanged.
In P_2, first **receive** $C_A(a)$ and **receive** $C_B(b)$ are exchanged
In P_2, then **receive** $C_C(c)$ and **receive** $C_B(b)$ are exchanged

This results in the resources being requested strictly in the order $B \rightarrow C \rightarrow A$ by all three processes. Thereby the system is *deadlock free* according to the resource ordering principle.

[In the given system, the request ordering of A and C in processes P_1 and P_2 does not actually matter if they start requesting B. If B has been acquired, only one of the processes can request further resources and then there will be enough A and C instances for that process as well as for P_3.]