NOTE: Maximum grade for questions 1, 2 is 10p, and for questions 3, 4 and 5 it is 6p.

1. Each of the three arrays `A[]`, `B[]` and `C[]` contains an ordered (ascending) sequence of integers. It's know that at least one integer value appears in all of the three sequences. Consider the following concurrent program aimed to find the smallest of such common values, X say:

```
int  a = A[0]; b = B[0]; c = C[0]; i,j,k = 0,0,0;
co   while true if (a < b) {i = i+1; a = A[i]};
//   while true if (b < c) {j = j+1; b = B[j]};
//   while true if (c < a) {k = k+1; c = C[k]};
oc
```

(a) Assuming first, that the scheduler executes each of the `if`–statements of three loop bodies as atomic actions, i.e. sequentially, show that the following assertion is an invariant of the program:

I: {(For all $i' \le i$: $A[i'] \le X$) and (For all $j' \le j$: $B[j'] \le X$) and (For all $k' \le k$: $C[k'] \le X$)}

and based on it, make argument that the program works correctly, i.e. the state {a = b = c = X} will be reached, after which there is no progress. (b) Does the program meet the requirements of the At-Most-Once property as a concurrent program, where each reference to any variable is an individual atomic action? (c) Show that I holds also then. (d) The program misses to terminate itself properly, why ? (e) Show how to fix it, and give an argument showing that the termination doesn't happen prematurely.

2. Consider the following proposal for semaphore implementation, where INC(x) atomically adds, and DEC(x) atomically subtracts 1 from integer variable x. Both INC(x) and DEC(x) return the new value of x:

```
/* P(s): */
while (DEC(s) < 0) {
    INC(s);  #undo decrement
    }

/* V(s): */
INC (s)
```

Answer to the following questions with a proper argumentation.
(a) Is this solution safe ? Provide and argumentation showing that the standard semaphore invariant : {value(s) = initial(s) + #V(s) - #P(s) ≥ 0 }, is preserved. The #V(s), #P(s) denote the number of *passed* V(s) and P(s) operations.
(b) Is there a possibility for a deadlock ?
(c) Does it exclude starvation, i.e. is the solution fair assuming normal weakly fair scheduling ?
(d) Is there a possibility for livelock ?
(e) What is bad with it performance wise?
(f) How would you improve its performance?
(g) Analyze your improved version also against your answers from (a) to (d). Is there any change ?

3. Give all possible final values and corresponding traces of the variable x leading to proper termination of the following program. Assume that all the assignment statements to x are executed as atomic actions.

```
int x = 0; sem s1 = 1, s2 = 1, s3 = 0;

co P(s1); x = x + 7; P(s2); x = 2 * x; V(s3); x = x * 7; V(s1);
// P(s1); x = x + 5; V(s2); x = 5 * x; P(s3); x = x * 13; V(s1);
// P(s2); x = x + 3; P(s1); x = x + 9; V(s2); x = x * 11; V(s1);
oc
```

4. One-lane bridge. Cars coming form north and south have to cross a river along a very long and narrow one-lane bridge. Cars driving to the same direction may be on the bridge at the same time, but cars heading to opposite directions can't. Consider the following generic monitor code outline for the solution to the problem, where the cars are processes calling the public methods cross_from_North() and cross_from_South() of the monitor One_lane_bridge.

```
monitor One_lane_bridge {
    int ns =0;              //north-south cars on the bridge
    int sn =0;              //south-north cars on the bridge
    cond ns_c:              //condition to enter from north
    cond sn_c;              //condition to enter from south

private procedure startNorth() {
       …
       ns++
       }
private procedure endSouth() {
       ns--;
       …
       }

public cross_from_North() {  // this is needed to provide a simpler API
    startNorth();
    // north-south crossing operation is embedded here
    endSouth();
  }

 // the south-north direction is symmetric and need not be repeated
}
```

Complete the missing peaces of the synchronization logic of the monitor procedures `startNorth()` and `endSouth()` without fairness considerations for : a) generic monitor with SC-semantics, (b) with SW-semantics, (c) modify the solution to work with Java synchronized classes and methods. NOTE: The solution can and should be simple, short and clearly written. Any failure to meet these criteria will decrease the grading respectively.

5. Tuple Space. (a) Write a simple, straight-line (= loop-free) tuple space solution to the readers-writers problem using the tuple space primitives of Java: postnote, removenote, and readnote. Hint: Use just a simple tuple ("RW", r) maintaining the number of active readers. Each of the synchronization operations enter-read, exit-read, enter-write, exit-write should take very few tuple operations. (b) Is your solution fair ? NOTE: The solution can and should be simple, short and clearly written. Any failure to meet these criteria will decrease the grading respectively.