T-106.5600 Concurrent Programming Examination December 2012 / hsa

Be concise and clear. The shorter you can be, the better.

1. Three ascending integer sequences A[0..] , B[0..] and C[0..] contain at least one common value. Consider the following program consisting of tree concurrent threads aimed to find the smallest of such values, X say, to reach the final state R: {a = b = c = X} , where it should terminate.

```
int a,c,b;
a = A[0]; b = B[0]; c = C[0]; i= j= k= 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
```

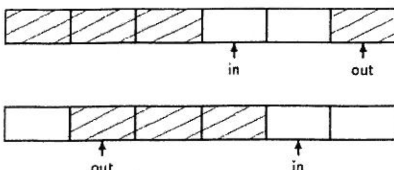
a) Let's assume, that the `if`–statements in loop bodies are executed as atomic actions. Give an argument with a proper invariant I that the program progresses and reaches the state R: {a = b = c = X}.
b) Assuming that each individual reference to the shared variables a, b and c is atomic. Explain whether or not I still an invariant of the program.
c) The program doesn't terminate properly. Show how to enhance it for proper termination, and explain whether or not R holds at the termination of your enhanced version both version a) and b).

2. Consider the simple bakery algorithm for two-process critical section:

Algorithm 5.1: Bakery algorithm (two processes)			
integer np ← 0, nq ← 0			
	p		q
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	np ← nq + 1	q2:	nq ← np + 1
p3:	await nq = 0 or np ≤ nq	q3:	await np = 0 or nq < np
p4:	critical section	q4:	critical section
p5:	np ← 0	q5:	nq ← 0

a) Show the safety with a proper invariant.
b) Find a scenario where the mutual exclusion does not hold when the statement p2 (symmetrically for q2) is replaced by two statements "p2.1: temp <- nq" and "p2.2: np <- temp +1".
c) Give an argument using temporal logic whether the algorithm is fair, i.e. neither p or q will starve or not.

3. A bounded buffer is frequently implemented using a circular buffer, which is an array that is indexed modulo its length:



One variable, *in*, contains the index of the first empty space and another, *out*, the index of the first full space. If *in* > *out*, there is data in *buffer*[*out*..*in*-1]; if *in* < *out*, there is data in *buffer*[*out*..*N*] and *buffer*[0..*in*-1]; if *in* = *out*, the buffer is empty. a) Prove that following algorithm 6.19 for the producer-consumer problem works correctly, i.e. it will not cause under or overflow of the buffer. b) Will it work with multiple producers and consumers? Justify your answer! If not, how should it be changed to make it work?

Algorithm 6.19: Producer-consumer (circular buffer)			
	dataType array [0..N] buffer integer in, out ← 0 semaphore notEmpty ← (0,∅) semaphore notFull ← (N,∅)		
	producer		consumer
	dataType d		dataType d
	loop forever		loop forever
p1:	d ← produce	q1:	wait(notEmpty)
p2:	wait(notFull)	q2:	d ← buffer[out]
p3:	buffer[in] ← d	q3:	out ← (out+1) modulo N
p4:	in ← (in+1) modulo N	q4:	signal(notFull)
p5:	signal(notEmpty)	q5:	consume(d)

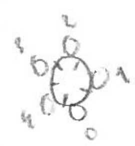
Algorithm erratum: in p4 and q3, "modulo N" should be "modulo (N+1)". Note that the algorithm considers the buffer to be full when one element in the array is unused.

4. One-lane bridge. Cars coming from 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 monitor solution `One_lane_bridge` to the problem, where the cars are processes calling the public methods `cross_from_North()` and `cross_from_South()`.

```
monitor One_lane_bridge {
int ns =0;            //north-south cars on the bridge
int sn =0;            //south-north cars on the brigde
cond ns_c:            //condition to enter from north
cond sn_c;            //condition to enter from south
private procedure startNorth() {
      if (sn > 0) wait(ns_c);
      ns++
      }
private procedure endSouth() {
      ns--;
      if (ns == 0) signal_all(sn_c));    //signals possible waiting sn cars
      }
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
```

a) Show by using proper invariants that the solution is safe and does not cause any unnecessary waiting.
b) Transform the example code in question 4 to a similar Java version.



5. Solve the five dining philosophers problem using tuple-space, so that the utilization rate of the forks is maximal, i.e. a philosopher don't reserve any forks until both of them are available after which he starts eating immediately. Hint: If a philosopher can't start eating, he goes into an explicit waiting state "hungry", so that when either of his neighbours leaves eating, he can be awakened to eat if both of his forks are available. Define clearly the meaning of the different tuple types and elements used in them and attach appropriate tags to them. The get_to_eat() and end_eat() operations should be simple algorithms without loops and using no other global variables except tuples.

Linda tuple-space primitives are: `postnote ('tag', ...)`, `readnote ('tag', ...)`, `removenote ('tag',...)`. Indicate clearly in a `readnote(...)`, or `removenote(...)` operation when a matching tuple with an element value equal to a program variable value is sought for (syntax " v="), from the case where an element value of a otherwise matching tuple is just assigned to a program variable (syntax " v").