

Answers for the exam of Wednesday, 9 January 2019  
**Concurrent Programming TDA383/DIT390, and**  
**Principles of Concurrent Programming TDA384/DIT391**

- Q1. (Part a)** No, there may be no output: q completes, then p runs. (2p)  
**(Part b)** Once. After writing 2, p1 fails test, so no more output. (2p)  
**(Part c)** Fair: A finite number of times. Do q1, then run p many times. Then q2 must run, which makes p exit. (2p)  
**(Part d)** unfair -> infinitely often. Do q1, then always run p. (2p)

**Q2. Part a** (4p)

$s = (p_i, q_i, \text{token}, t_p, t_q)$	$sp = \text{next state if p moves}$	$sq = \text{next state if q moves}$		
s3	(p2, q3, _, _, Q)	(p3, q3, _, _, Q) = s8	(p2, q5, _, _, Q) = s4	
s4	(p2, q5, _, _, Q)	(p3, q5, _, _, Q) = s9	(p2, q2, T, _, _) = s1	
s5	(p3, q2, T, _, _)	(p2, q2, T, _, _) = s1	(p3, q3, _, _, Q) = s8	
s8	(p3, q3, _, _, Q)	(p2, q3, _, _, Q) = s3	(p3, q5, _, _, Q) = s9	
s9	(p3, q5, _, _, Q)	(p2, q5, _, _, Q) = s4	(p3, q2, T, _, _) = s5	
Ans:	s1	(p2, q2, T, _, _)	(p3, q2, _, P, _) = s6	(p2, q3, _, _, Q) = s3
	s2	(p2, q3, T, _, _)	(p3, q3, _, P, _) = s7	(p2, q2, T, _, _) = s1
	s6	(p3, q2, _, P, _)	(p5, q2, _, P, _) = s10	(p3, q3, _, P, _) = s7
	s7	(p3, q3, _, P, _)	(p5, q3, _, P, _) = s11	(p3, q2, _, P, _) = s6
	s10	(p5, q2, _, P, _)	(p2, q2, T, _, _) = s1	(p5, q3, _, P, _) = s11
	s11	(p5, q3, _, P, _)	(p2, q3, T, _, _) = s2	(p5, q2, _, P, _) = s10

- (Part b)** There is no (p5, q5, ...) state. (1p)  
**(Part c)** No, "no moves". Only busy waiting. (1p)  
**(Part d)** Yes, X shows where the token is: with P, Q, or neither. (3p)  
**(Part e)** Interleave s1, s3, s4 cycle with a run of p2; p3 after q has acquired the token. (4p)  
**(Part f)**  $S_1 = \{s6, s7\}$  is the smallest set that satisfies the condition. The largest would be the set of all the states in the system. (3p)

**Q3.** Let  $I$  be  $(t_p + t_q + \text{token} = 1)$ , and  $M$  be  $\neg(p5 \wedge q5)$ .

- (Part a).** At init, token=1,  $t_p=t_q=0$ . Only swap changes these variables, preserves I. (2p)  
**(Part b).** p5 can only be reached by  $t_p=1$  at p3. p5 and q5 cannot happen because both  $t_p$  and  $t_q$  cannot be 1. (2p)  
**(Part c).** p5 or q5 implies token=0. Contrapositive. (2p)

**(Part d).** to show  $(p2 \wedge q2) \rightarrow (\text{token} = 1)$ , first show that  $q2 \rightarrow (\text{tq} = 0)$ . This is true at init, and q can only return to q2 via q3 or q5, in both cases with tq=0. Similarly for p2. (2p)

**(Part e).** If token=1, then neither p5 nor q5. One of them, say p, will do a swap first, get the token. If token=0 then tp=1 (say). Wherever p is, it will do a swap, and token=1. (3p)

**Q4. (Part a).** One of the eb's, then a wb, then the other eb, and another wb. One wb remains. (5p).

**(Part b).** The marooned wb can be rescued by a shuttle crossing eastwards. But the shuttles might loop, monopolising the bridge. (4p).

**(Part c).** Sleep would make the shuttles of lower priority than the cars. (3p)

**(Part d).** Initialize the semaphores to a sum of 3. to put 3 cars on the bridge. Use shuttles as needed to get stranded cars across. (3p)

**Q5. (Part a).** What output does `go([])` give? Ans: None. (2p).

**(Part b).** What output does `go([5])` give? Ans: "5, ". (2p).

**(Part c).** What output does `go([6, 5])` give? Ans: "5, 6, ". (4p).

**(Part d).** Output of `go([-3,5,4,103,4,0])` is "4, 4, 5, " (4p).

**Q6. for TDA383/DIT390**

**(Part a).** A Prod-Cons monitor, buffer size 1. Symmetric, so either can run first. Prod waits for Count==0 and sets Count=1. Cons is vice-versa. Off bridge car actions done by users of the monitor. (3p).

**(Part b).** The same Prod-Cons will do. Run dummy cars that use these, and make the bridge swap direction every so often. (2p).

**(Part c).** The dummy cars sleep between runs. (1p).

**(Part d).** Bridge entry and exit are in charge of the monitor, no risk of user forgetting. Use of monitor clearly separated from monitor – see parts b and c. Some loss of parallelism. (2p)

**Q6. for TDA384/DIT391**

**(Part a).** Save time on locking if rarely needed. (2p)

**(Part b).** If thread `t` removes a set element behind `u`'s curr, then `u` will include it in the count even if `t` terminates before `u`. (3p)

**(Part c).** (3p)

```
int curSize;
do {
    curSize = size.get();
} while (!size.compareAndSet(curSize, curSize + 1));
```