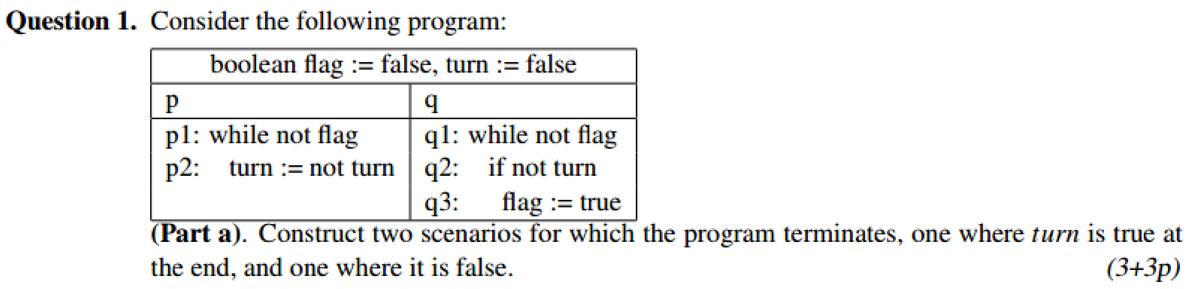


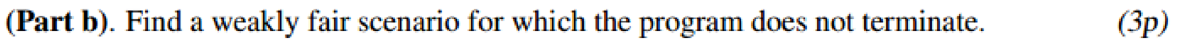
Question 1. Consider the following program:

boolean flag := false, turn := false	
p	q
p1: while not flag p2: turn := not turn	q1: while not flag q2: if not turn q3: flag := true

(Part a). Construct two scenarios for which the program terminates, one where *turn* is true at the end, and one where it is false. *(3+3p)*

scenario turn = true:
**Since there is no mutual exclusion we can just do:*
*p2**q2** (turn = false) → **p2**q3 (turn = false) → p1**q3** (turn = true) → **p1**q1 (flag = true) → (end)**q1** (flag = true) → (end)(end)*

scenario turn = false:
*p1**q2** (turn = false) → p1**q3** (turn = false) → p1**q1** (turn = false, flag = true) → **p1**(end) (flag = true) → (end)(end) (flag = true)*



(Part b). Find a weakly fair scenario for which the program does not terminate. *(3p)*

For this to work, we just need turn to be true whenever we execute q2. As such, we need to first run the whole of p once (so turn = true). We can then run q once. For the rest of the time,

we need to run p twice before running q again. So p will run twice as often, but q will never fully starve.
As a scenario:
***p1**q1 → **p2**q1 → p1**q1** → p1**q2** → *now we run p twice for every q* so (abbreviated): **p**q → **p**q → **q**p → **p**q → **p**q → etc*

Question 2. **(Part a)**. Using synchronous channels, develop a program to sort n numbers, where $1 \leq n \leq 100$. Assume that the numbers to be sorted are all distinct, positive, non-zero integers. The n numbers are fed into a channel c_0, and a sentinel value 0 is fed into c_0 to signal the end of input.

Build a chain of n processes P_i, for $1 \leq i \leq n$, and channels c_i for $0 \leq i \leq n$. Each process P_i has channel c_{i-1} to its left and channel c_i to its right. Process P_i takes input from channel c_{i-1} and delivers output via channel c_i to process P_{i+1} to its right. Process P_1 takes from c_0 the input numbers to be sorted. If you need it, write a sink process to input numbers from c_n and throw them away.

When the program terminates, the input numbers should be stored one per process in local memory. Let d_i be the number held by process P_i. Then for i and j such that $i < j$, it should be that $d_i < d_j$. Write the code for the processes P_i to sort the input as described. *(8p)*

** This kind of works like an access table implemented with a linked list…*
** Since we have an index for all possible values, we can just send the values forth until they're in their "correct" place. This hurt me right in the algorithm…*

```
------------------------------------------------------------------------------------
channel of integer[0….. n] c          *c0 corresponds to c[0]
-------------------------------------------
(process) sorter
integer index := i
integer number, sortedNumber
-----------------------
loop
p1  number <= c[i-1]
p2 if number > index  then
p3       number => c[i]
p4   else if number = 0
p5       number => c[i]
p6       break
p7  else  sortedNumber := number
```

(Part b). Adapt your program to work with asynchronous channels. Assume the buffering capacity of each channel is 100. *(3p)*

**Since the buffer will not overflow with capacity 100, we don't need a guard on whether the buffer is full*
Probably, we don't need to change anything, since the process will end automagically when it receives 0 as input, and the input is still received in the same order so it should not make any difference…

Question 3. Here is yet another algorithm to solve the critical section problem, built from atomic "if" statements (p2, q2 and p5, q5). The test of the condition following 'if', and the corresponding "then" or "else" action, are both carried out in one step, which the other process cannot interrupt.

integer S := 0	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: if even(S) then S:=4 else S:=5	q2: if S $<$ 4 then S:=3 else S:=7
p3: await (S$\neq$ 1 $\wedge$ S$\neq$5)	q3: await (S$\neq$ 6 $\wedge$ S$\neq$7)
p4: critical section	q4: critical section
p5: if S $\geq$4 then S:=S-4 else skip	q5: if odd(S) then S:=S-1 else skip

Below is part of the state transition table for an abbreviated version of this program, skipping p1, p4, q1 and q4 (the critical and non-critical sections).

A state transition table is a tabular version of a state diagram. The left hand column lists the states (where *p* and *q* are, and the value of *S*). The middle column gives the next state if *p* next executes a step, and the last column gives the next state if *q* next executes a step. In many states both *p* or *q* are free to execute the next step, and either may do so. But in some states, such as 5 below, one or other of the processes may be blocked. There are 10 states in all.

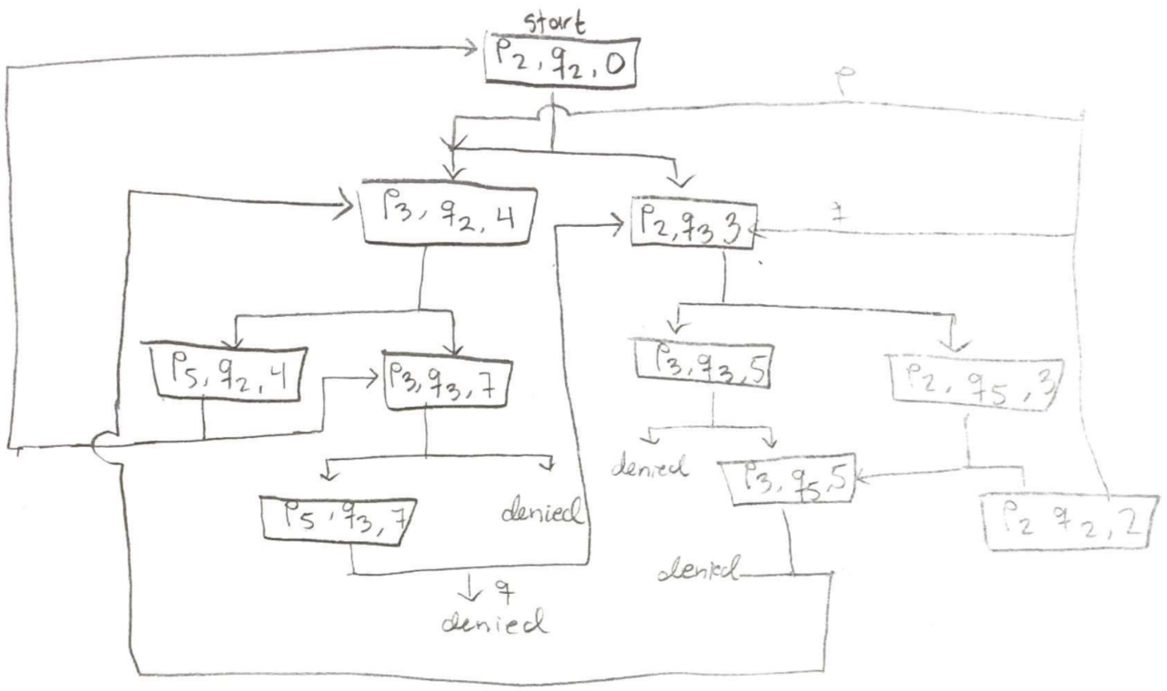
	State = (pi, qi, Svalue)	next state if p moves	next state if q moves
1.	(p2, q2, 0)	(p3, q2, 4)	(p2, q3, 3)
2.	(p3, q2, 4)	(p5, q2, 4)	(p3, q3, 7)
3.	–	–	–
4.	–	–	–
5.	(p3, q3, 7)	(p5, q3, 7)	no move
6.	–	–	–
7.	–	–	–
8.	–	–	–
9.	–	–	–
10.	(p2, q2, 2)	(p3, q2, 4)	(p2, q3, 3)

(Part a) Complete the state transition table. *(6p)*

Solution:

	State = (pi, qi, Svalue)	next state if p moves	next state if q moves
1.	(p2, q2, 0)	(p3, q2, 4)	(p2, q3, 3)
2.	(p3, q2, 4)	(p5, q2, 4)	(p3, q3, 7)
3.	(p5, q2, 4)	(p2, q2, 0)	(p3, q3, 7)
4.	(p5, q3, 7)	(p2, q3, 3)	no move
5.	(p3, q3, 7)	(p5, q3, 7)	no move
6.	(p2, q3, 3)	(p3, q3, 5)	(p2, q5, 3)
7.	(p3, q3, 5)	no move	(p3, q5, 5)
8.	(p3, q5, 5)	no move	(p3, q2, 4)
9.	(p2, q5, 3)	(p3, q5, 5)	(p2, q2, 2)
10.	(p2, q2, 2)	(p3, q2, 4)	(p2, q3, 3)

**In a more understandable tree-form:*



*A tip is to draw such a diagram as this before filling in the transition table. Then you'll better understand **why** there are only ten states and how they're interconnected.*

(Part b) Prove from your state transition table that the program ensures mutual exclusion. *(2p)*

Because in the diagram, there are no instances where both p and q can be in q5, at least one will be blocked in state 3 (p3/q3)

(Part c) Prove from your state transition table that the program does not deadlock (there are await statements, so it is possible for a process to block). *(2p)*

Because there are no instances where both processes cannot make a move, as seen on the transition table, and on the graph above. Therefore there can never be a deadlock.

Question 4. Refer again to the program in Question 3. This time, you must argue from the program, not from the state transition table (though you may seek inspiration from it!).

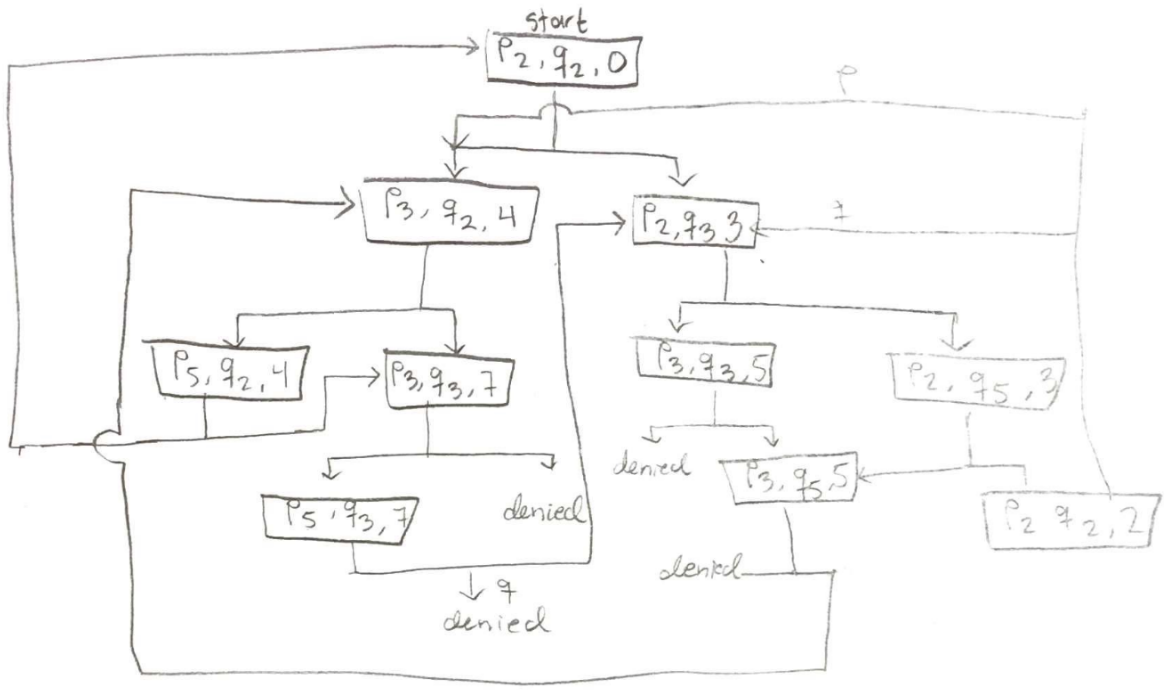
(Part a). Show that $(p3 \wedge q3) \rightarrow (S = 5 \vee S = 7)$ is invariant. *Hint:* Reason about what must have happened for the program to get to $(p3 \wedge q3)$. *(4p)*

This invariant concludes that if we're in the state p3 ^ q3, then that must mean S can only take on the values 5 or 7. We can prove this by showing that there is no state p3 ^ q3 where S equals anything else. we have :
det var väl inte helt som jag trodde; det var typ bara några killar som håller på med ett projekt. Men de hade helt okej planer på hur det skulle bli, och de har redan flera andra företag som de finansierar det med.

(p2, q3, 3) → (p3, q3, 5)
(p3, q2, 4) → (p3, q3, 7)

Because S will always be 3 from state p2,q3, it will, given p2-p3 is the next move, since 3 is uneven, it will always turn into 5 the next round.

Because S will always be 4 whenever we are in p3,q2, and q2-q3 makes S 7 whenever it's greater than 3 in q2, we will only ever get this result.



We can also see this in the diagram above showing the connection between all states, as we can’t get to a state before p3q3 where the value of S is not either 4 or three with corresponding states.

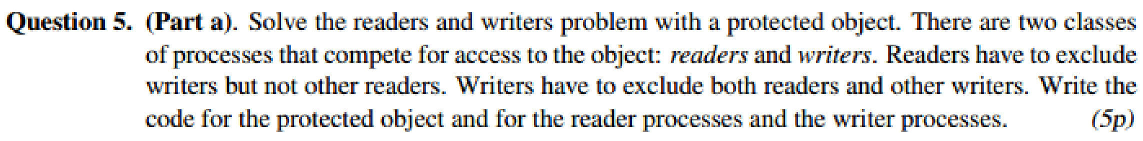
(Part b) Assume that $(p3 \wedge q5) \rightarrow$ (S=5). Prove that if $p3 \wedge q5$, then p cannot move until after q executes $q5$. (That is, mutual exclusion holds). *(2p)*

Because p3 will be blocked until S is not 5 or 1 (and it can’t be 1 ever anyway)

(Part c) Assume that $p3 \wedge q1 \rightarrow (S = 4)$ is invariant, and that q always loops in q1. Then prove that $p3 \wedge q1 \rightarrow \Box\Diamond p5$. *(4p)*

**We can assume S is four to begin with, and that q is only ever doing stuff in q1. Because of this, p can always loop through its whole sequence (since S will always be even -> not become 5 and be blocked in q3) and thus it is always possible for p to enter p5.*

**So S will only toggle between 4 in p2 and 0 in p5. Therefore it will remain even and always be able to get to p5 again. As the dictum dictates, p3 ^ q1 → p5 will always at some point become p5 again, and when it isn’t in p5 it will get there again…*


Question 5. **(Part a).** Solve the readers and writers problem with a protected object. There are two classes of processes that compete for access to the object: *readers* and *writers*. Readers have to exclude writers but not other readers. Writers have to exclude both readers and other writers. Write the code for the protected object and for the reader processes and the writer processes. *(5p)*

protected object RW

```
    integer readers := 0
    boolean writing := false

    operation StartWrite when not writing and readers = 0
        writing := true

    operation EndWrite
        writing := false

    operation StartRead when not writing
        readers := readers +1

    operation StopRead
        readers := readers -1
```

reader

```
loop forevs
p1 StartRead
p2 read stuff
p3 EndRead
```

writer

```
q1 StartWrite
q2 write stuff
q3 EndWrite
```

----

reader

```
p1 StartRead()
p2 read database
p3 EndRead()
```

---

writer

q1 StartWrite()
q2 write stuff
q3 EndWrite()

(Part b). Suppose a process P is waiting on a barrier to an operation on the protected object. How does the system know when P should be unblocked? *(3p)*

**A barrier on a protected object is recalculated every time an operation on the object is completed.*
**A process blocked on a barrier is assigned a spot in a FIFO queue associated with that barrier. So the first process in the queue is unblocked when the barrier is fulfilled.*

Question 6. Consider the program below, with binary semaphores as the sole communication method.

binary semaphore SA:=0, SB := 0, SC := 1		
process A	process B	process C
loop forever	loop forever	loop forever
wait(SB);	wait(SC);	wait(SA)
print("A");	print("B");	print("C");
signal(SC)	signal(SA)	signal(SB)

(Part a). What does the program print? *(3p)*

**Since all processes except the one waiting for SC will be blocked initially; the one waiting on SC will run first.*
Since that is process B, we get a B.

**Process B signals SA, which process C is waiting for.*
process C prints C.

**Process C signals SB, which process A is waiting for*
Process A prints A

**Now process A signals SC again. Since this is exactly as it started, we will get the same prints over and over:*

BCA BCA BCA BCA BCA

(Part b). Show that when any process is printing, all three semaphores are zero. Where are the other two processes at that time? *(3p)*

To show this, I will look at the program in the sequence it is run. The program begins with only one mutex being available. When process B finishes the 'wait' statement, SC is decreased to zero. Only after printing, it signals another mutex which will then become 1 until the process waiting for that mutex aquires it. Therefore there will always be at most one available semaphore at any time.

(Part c). Suppose we had declared the semaphores to be general semaphores instead, but with the same initial values as above. Would the program printout be different? *(2p)*

**Since we initiate the program with only one semaphor at 1, the others at zero, we would have to signal the same process twice for the order of the prints to change. Since we do not use signal more than once in each process, this cannot happen.*

(Part d). The given program always initialises SC to 1 and the other semaphores to zero. Suppose we would like to initialise all the semaphores to zero, and then randomly signal one of them to get the above program started. Write processes to run in parallel with A, B and C, to achieve this without a random number generator. You may use as many processes and semaphores as you like, but no other communication method. *(4p)*

**We want to create one process for signaling each semaphore. For them not to signal more than one, we define a mutex where the first one to grab it gets to signal its assigned semaphore.*

binary semaphore OKsignal := 1

process StartSA
p1 wait(OKsignal)
p2 signal (SA)

process StartSB
q1 wait(OKsignal)
q2 signal(SB)

process StartSC
r1 wait(OKsignal)
r2 signal(SC)

Question 7. **(Part a)**. Implement a bounded buffer of capacity n, where $n \geq 1$, in Linda. Assume the products to be stored in the buffer are all alike.

Write a producer process P and a consumer process C. The producer puts products v into the buffer, and has to wait if (and only if) the buffer is full—that is, when there are n products in the buffer. The consumer takes products from the buffer, and has to wait if (and only if) the buffer is empty—that is, when there are no products in it.

Your program will get most credit if the processes wait *only* for the conditions described above, and if the processes P and C maintain as little internal information as possible. But you can post as much other information as you need into the space. *(7p)*

**We think like this: we have two classes, producers and consumers. Somehow we have to ensure there can only be n objects in the buffer, and we don't want the producer/consumer classes to hold too much information. We could implement this with a notFull/notEmpty with a count like in the book, however then processes will have to wait which they shouldnt!*

**So we implement it by assuming there are n "empty" objects in the space, which the producer then 'fills' and puts back with another pattern match which the consumer waits for. This way the buffer will start full, but never exceed max. It will always contain the same amount of nodes, however the amount 'Empty' and 'Product' may vary as the progam runs.*

producer P

```
productType product
-------------
loop forevs
p1 product := produce
p2 removenote('Empty')
p2 postnote('Product', product)
```

consumer C

```
productType product
-------------
loop forevs
c1 removenote('Product', product)
c2 consume(product)
c3 postnote('Empty')
```

(Part b). Generalise your program to allow multiple instances of process P and multiple instances of process Q. That is, to allow multiple producers (all running the same code P) and multiple consumers (all running the same code C). *(3p)*

-

——-END of QUESTION PAPER——-

**The current program works for multiple processes, so it shouldn't be a problem!*