HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Benjamín Aage B. Birgisson

Exercise 4.1.2:

(a)

The expected value would be 6 (with sequential execution), as p starts executing with adding 1+0 and storing that result of x, then q takes over and loads the value of x and executes 1+1, resulting in x being 2 and stores that value. In the next round of execution, p comes back and loads the value of x and then executes 2+1, storing the value of x being 3, and then q takes over and executes 3+1. Third and final round, p begins loading the value of x (being 4 here) and executes 4+1, stores the value of x, and afterwards q loads the value of x and then executes 5+1, making the final value of x=6.

(b)

For example, if p would load the value of x (0), then q would load the value of x (0) as well, then p would increment x by one, and then q would increment x by 1 and store the value of x (1). Afterwards, p would finally store the value of x (1). So, with this looping for three times, the final value of x would become 3.

(c)

Same as in section b here above, the value of x would equal 3, as q would overwrite the value of x being given by the process of p.

Exercise 4.2.1:

(a)

p1:	p2:	p3:	p4:
P(s) A CS V(s) P(s) B CS V(s) P(s) C CS V(s) P(s) D CS	p(s) E CS V(s) P(s) F CS V(s)	P(s) G CS V(s)	p(s) H CS V(s)

s=1 is the starting point, then A can complete with P(s), making s=0. Then A produces V(s), so s=1, then E can operate, making s=0.

Afterwards, E produces V(s), making s=1, so that H can operate with P(s), making s=0 again. E produces V(s), making s=1.

Then, B can operate with P(s), and then produce V(s), keeping s=1.
Next is F, which does the same, keeping s=1.
Then G, after that C, and at then very end D (making s=0).

(b)

s = 1, so that A can start operating and complete.
A single semaphore initialized to 1 is sufficient enough to solve the problem for any number of processes, as is the case here.

Exercise 4.2.3:

(a)

A : 0	→ P(s1) →	A : 0	→ P(1) →	V(s1) →	A : 1
s1 : 1		s1 : 1			s1 : 2
c1 : 0		c1 : 1			c1 : 1
d : 1		d : 1			d : 0

A could invocate two times, as s1 equals two.
The value of s1 = 2, c1 = 1 and d = 0.

(b)

B : 0	→ P(s2) →	B : 0	→ V(s2) →	B : 1
s2 : 1		s2 : 0		s2 : 1
c2 : 0		c2 : 1		c2 : 2
d : 1		d : 1		d : 0

B could invocate one time, as s2 is equal to one.
The value of s2 = 1, c2 = 2, and d = 0.

(c)

No, because one of the blocked processes is chosen at random and enters the CS. Process i may continue but starvation is not possible since process I cannot re-enter CS before the chosen process currently running finishes.

Exercise 4.3.1:

(a)

Pb(sb) : sb = 1
Vb(sb) : do SWAP(R,x) while R == 0

(b)

Pb(sb) : sb = 1
Vb(sb) : do TSB(x,L) while x == 0

Exercise 4.4.1:

(a)

-	Original state:	X = 10	Y = 2	
-	m.A();	X = 11	Y = 9	
-	m.B();	X = 10	Y = 9	
-	m.B();	X = 10	Y = 9	~~c.wait~~
-	m.B();	X = 10	Y = 9	~~c.wait~~
-	m.B();	X = 10	Y = 9	c.wait
-	m.A();	X = 11	Y = 9	c.signal
		X = 10	Y = 8	
-	m.A();	X = 11	Y = 9	c.signal
		X = 10	Y = 8	

Exercise 4.4.4:

(a)

```
f(x) {
    P(mutex)
    if (x) {
            wait_count = wait_count + 1
            c1.wait
            }
    x = x + 1
    V(mutex)
    c2.signal
    x = 0
    }
```

Exercise 4.4.6:

(a)

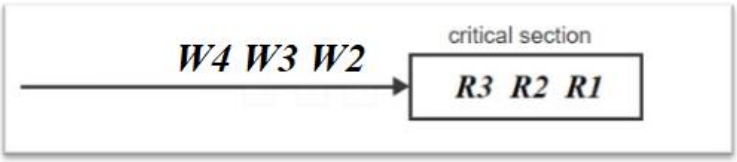
As soon as function g() has been called, x will never become negative again. This could result in starvation of process(es).

(b)

Activate a block (by some condition preferred) on the urgent queue, to block the invoking process.

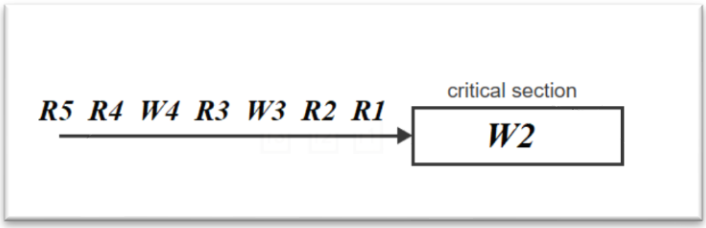
Exercise 4.5.1:

(a)



When all the readers have finished, the next writer can be processed (W2), and when that has finished, the next writer can be processed as long as there has no readers entered the queue in the meantime. Same goes after writer 3 has finished, then writer 4 can be processed as long as there has no additional readers entered the queue.

(b)



When all of the readers have finished, W3 can be processed in the critical section, and at last W4.

Exercise 4.5.3:

(a)

There should be a counter for both reading and writing, but here it is only a counter for reading. There is also no ready-queue for both reading and writing, which are used to block readers and writers when needed.

This solution could lead to starvation of readers, as there is no mutex within the function of writer() to indicate that the slot is available for reader.

Exercise 4.5.6:

(a)

You can call ‘think’ which requires no resources. If a philosopher wants to eat, he must request a ‚fork‘ resource both on his left and on his right, which is done by the calling of P(mutex). By calling this, he must have both the left and the right fork, else he is not able to eat. If they are available, he can eat by calling of V(mutex) and afterwards release the resource by calling V(). So yes, in my opinion the solution satisfies all requirements of the philosophers problem. The only possible problem is perhaps that the philosopher on the left and right of the philosopher eating is not blocked, because they only have one available fork – but I think that this is satisfied by the P(f[i]) and P([i+1mod 5]) functionality as the philosopher has to request both forks at the same time to be able to eat.

```
p(i) {
  while (1) {
    think
    P(f[min(i, i+1 mod 5)])
    P(f[max(i, i+1 mod 5)])
    eat
    V(f[i])
    V(f[i+1 mod 5])
  }
}
```

Exercise 4.5.8:

a) If the destination is the same as the position of the elevator is, it will be ignored regardless of what the direction of the elevator is. So, if the direction is currently ‘up’, the destination will not be done until it goes back down again, making the travelling distance much more than necessary.

b) The condition would be the same before, except that the direction of the elevator is being ignored, causing more travelling distance between request.
