CSSE7610 – Assignment 2

Question 1

Part A

Below is a scenario in which two writers are implemented and the outcome of the reader does not behave as expected. In the following case, if two writers enter **q3** simultaneously, that the value of **c** will be even and therefore the reader may read **x1** and **x2** incorrectly.
Note: We will Assume values for **get()**

n	Reader	Writer 1	Writer 2	c	x1	x2
1	p1	**q1**	q1	0	0	0
2	p1	**q2**	q1	0	0	0
3	p1	**q3**	q1	0	0	0
4	p1	q4	**q1**	1	0	0
5	p1	q4	**q2**	1	0	0
6	p1	q4	**q3**	1	0	0
7	**p1**	q4	q4	2	0	0
8	**p2**	q4	q4	2	0	0
9	**p3**	q4	q4	2	0	0
10	**p4**	q4	q4	2	0	0
11	p5	**q4**	q4	2	0	0
12	p5	**q5**	q4	2	5	5
13	p5	q6	**q4**	2	5	5
14	**p6**	q6	q5	2	10	5
15	**p7**	q6	q5	2	10	5
16	**p8**	q6	q5	2	10	5

In the scenario above, when two writers increment **c**, the value of **c** will be even and allow the reader to read. This is an issue if the writers change the values of **x1**, and **x2** separately. In the case above we see that Writer 1 and Writer 2 increment **c** at the same time allowing the reader to read. While the reader is reading, the values **x1** and **x2** are change separately and therefore the reader will read incorrectly.

To combat this error, we will use a semaphore, specifically a strong semaphore to ensure freedom from starvation. Below is an implementation of the strong semaphore using the given algorithm.

Non-blocking reader-writer			
integer c, x1, x2 ← 0 binary semaphore S ← (1, ∅)			
	reader		**writer**
	integer c0, d1, d2		integer d1, d2
	loop forever		loop forever
p1:	repeat	q1:	d1 ← get()
p2:	repeat	q2:	d2 ← get()
p3:	c0 ← c	q3:	**wait(S)**
p4:	until (c0 mod 2 = 0)	q4:	c ← c+1
p5:	d1 ← x1	q5:	x1 ← d1
p6:	d2 ← x2	q6:	x2 ← d2
p7:	until (c0 = c)	q7:	c ← c+1
p8:	use(d1,d2)	q8:	**signal(S)**

By using a semaphore, we can ensure that the variables **x1** and **x2** are only ever being changed by one writer at a time and thus ensuring that a reader will only read the values from one writer at a time. If a writer executes **q3** then no other writer will be able to write and will have to wait until notified.

Part B

Below is the implementation of the original algorithm with the incrementor included.

Non-blocking reader-writer					
integer c, x1, x2 ← 0					
reader		**writer**		**incrementor**	
	integer c0, d1, d2		integer d1, d2		integer c0, d1, d2
	loop forever		loop forever		loop forever
p1:	repeat	q1:	d1 ← get()	q1:	repeat
p2:	repeat	q2:	d2 ← get()	q2:	repeat
p3:	c0 ← c	q3:	c ← c+1	q3:	c0 ← c
p4:	until (c0 mod 2 = 0)	q4:	x1 ← d1	q4:	until (c0 mod 2 = 0)
p5:	d1 ← x1	q5:	x2 ← d2	q5:	d1 ← x1
p6:	d2 ← x2	q6:	c ← c+1	q6:	d2 ← x2
p7:	until (c0 = c)	q7:		q7:	until (c0 = c)
p8:	use(d1,d2)	q8:		q8:	c ← c+1
p9:		q9:		q9:	x1 ← d1 +1
p10:		q10:		q10:	x2 ← d2 +1
p11:		q11:		q11:	c ← c+1

The incrementor will be a combination of both the reader and the writer process. This consists of a non-blocking reading part of the incrementor and a blocking writing/incrementing section.

For the incrementor to be low priority and to include multiple readers, writers, and incrementors we will make use of a monitor that will give priority to writers and ensure that readers do not block writing or incrementing.

Below is the above algorithm with a monitor implementation:

Non-blocking reader-writer

```
integer c, x1, x2 ← 0

monitor RW
    integer writers ← 0
    integer incrementors ← 0
    condition OKtoWrite, OKtoIncrement
    operation StartWrite
        if writers ≠ 0 or readers ≠ 0
            waitC(OktoWrite)
        writers ← writers + 1
    operation EndWrite
        writers ← writers - 1
        if empty(OktoIncrement)
        then signalC(OktoWrite)
        else signalC(OktoIncrement)
    operation StartIncrement
        if writers ≠ 0 or not empty(OktoWrite)
            waitC(OktoIncrement)
        incrementors ← incrementors + 1
        signalC(OktoIncrement)
    operation EndIncrement
        incrementors ← incrementors – 1
        if incrementers = 0
            waitC(OktoWrite)
```

reader		writer		incrementor	
	integer c0, d1, d2		integer d1, d2		integer c0, d1, d2
	loop forever		loop forever		loop forever
p1:	repeat	q1:	d1 ← get()	k1:	repeat
p2:	repeat	q2:	d2 ← get()	k2:	repeat
p3:	c0 ← c	q3:	**RW.StartWrite**	k3:	c0 ← c
p4:	until (c0 mod 2 = 0)	q4:	c ← c+1	k4:	until (c0 mod 2 = 0)
p5:	d1 ← x1	q5:	x1 ← d1	k5:	d1 ← x1
p6:	d2 ← x2	q6:	x2 ← d2	k6:	d2 ← x2
p7:	until (c0 = c)	q7:	c ← c+1	k7:	until (c0 = c)
p8:	use(d1,d2)	q8:	**RW.EndWrite**	k8:	**RW.StartIncrement**
p9:		q9:		k9:	if c0 == c
p10:		q10:		k10:	c ← c+1
p11:		q11:		k11:	x1 ← d1 +1
p12:		q12:		k12:	x2 ← d2 +1
p13:		q13:		k13:	c ← c+1
p14:		q14:		k14:	**RW.EndIncrement**
p15:		q15:		k15:	else
p16:		q16:		k16:	**RW.EndIncrement**

Here we can see that there will only ever be one incrementor or writer that can access the critical section at a time. The guarantees that the values of **x1** and **x2** can not be change by more than one thread at a time. By using condition variables, we can give priority to the writers over the incrementors and ensure that the incrementor does not block any writers during the reading section of the process. There is a possibility that between **k7** and **k8** that the values of **x1** and **x2** may have been changed so the incrementor will verify that there has been no change in **c** before it increments **x1** and **x2** otherwise it will end.