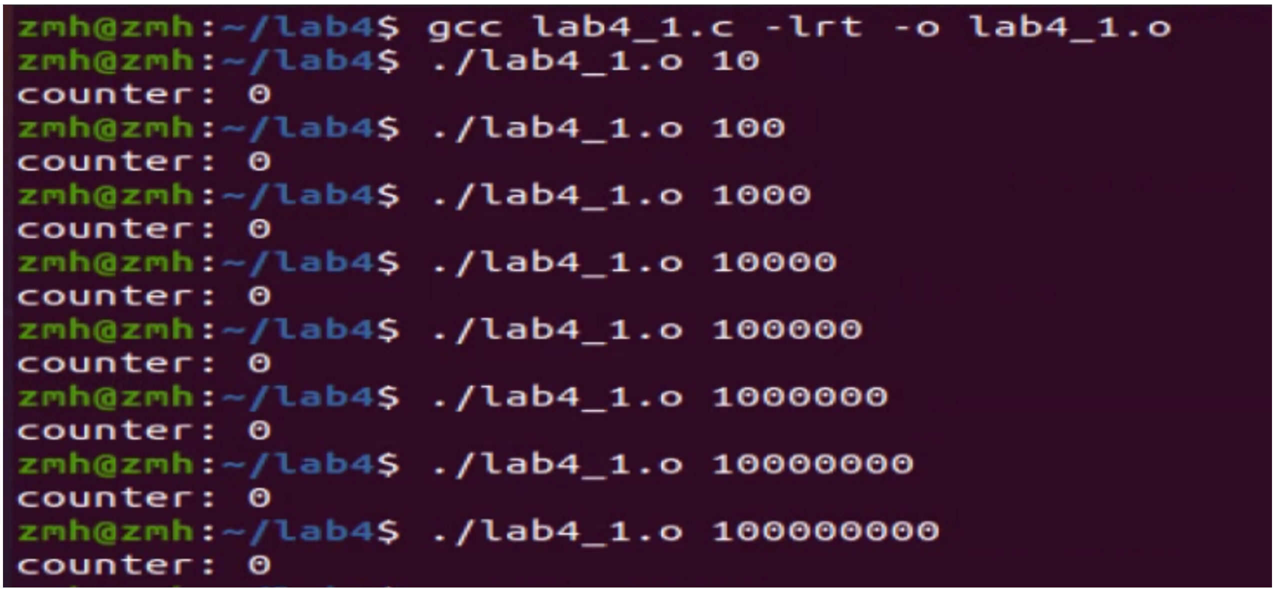
Lab4

problem 1

a. -lrt flag to use shm_open



```
zmh@zmh:~/lab4$ gcc lab4_1.c -lrt -o lab4_1.o
zmh@zmh:~/lab4$ ./lab4_1.o 10
counter: 0
zmh@zmh:~/lab4$ ./lab4_1.o 100
counter: 0
zmh@zmh:~/lab4$ ./lab4_1.o 1000
counter: 0
zmh@zmh:~/lab4$ ./lab4_1.o 10000
counter: 0
zmh@zmh:~/lab4$ ./lab4_1.o 100000
counter: 0
zmh@zmh:~/lab4$ ./lab4_1.o 1000000
counter: 0
zmh@zmh:~/lab4$ ./lab4_1.o 10000000
counter: 0
zmh@zmh:~/lab4$ ./lab4_1.o 100000000
counter: 0
```

b.

with atomic instructions

n	Counter
10	0
100	0
100,0	0
100,00	0
100,000	0
100,000,0	0
100,000,00	0
100,000,000	0

c.

Answer: It is a constant, the value is 0.

Reason:

I have tried two version of add and sub instructions. In the first version, I use the following code:

```
*counter = (*counter) + 1; //none atomic
*counter = (*counter) - 1; //none atomic
```

this is a non-atomic instruction which may lead the final value to be non-zero.

But in the second version, I tried to use the following sentence to add and sub

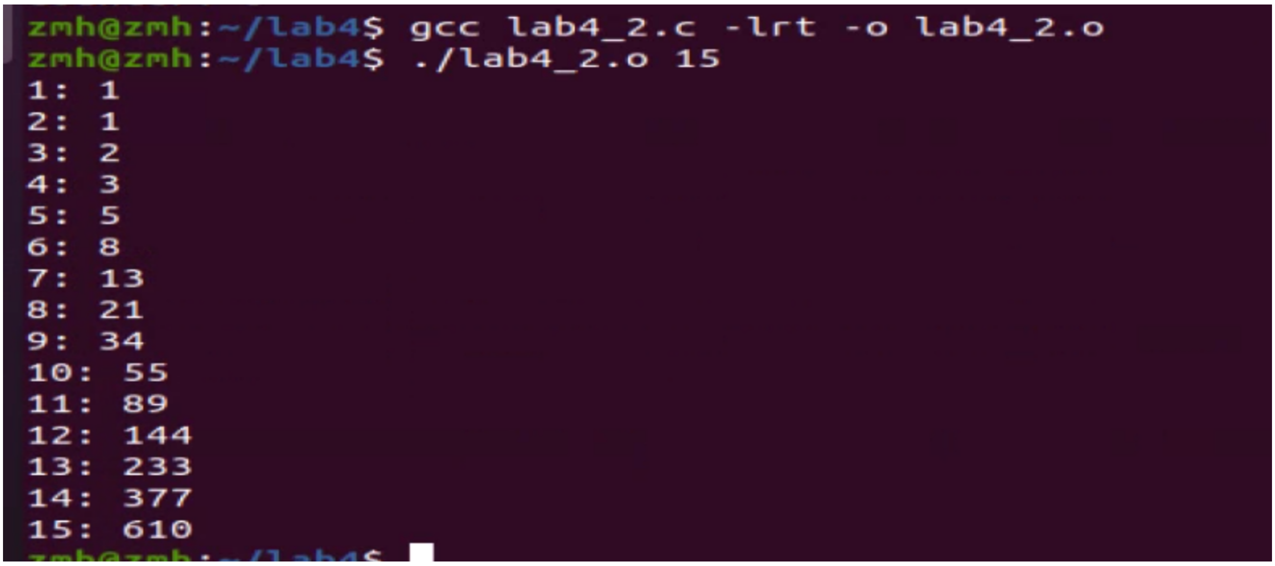
```
__sync_fetch_and_add(counter, 1);
__sync_fetch_and_sub(counter, 1);
```

Since they are atomic instructions, the counter will be add and sub the same time, so the result will always be zero.

d. No, when I use the atomic instructions, behaviors are always the same.

problem 2

a.



```
zmh@zmh:~/lab4$ gcc lab4_2.c -lrt -o lab4_2.o
zmh@zmh:~/lab4$ ./lab4_2.o 15
1:  1
2:  1
3:  2
4:  3
5:  5
6:  8
7:  13
8:  21
9:  34
10:  55
11:  89
12:  144
13:  233
14:  377
15:  610
zmh@zmh:~/lab4$ 
```

b. Using you implementation (as described in slides 5-8), what’s the maximum number of elements the shared buffer can actually hold? Why?

5*siezof(int) = 20B, the shared buffer can actually hold 20byte of elements. And it is a bounded buffer, so the maximum size will be only 20B.