

Lab4

problem 1

a. -lrt flag to use shm_open

```
zmh@zmf:~/lab4$ gcc lab4_1.c -lrt -o lab4_1.o
zmf@zmf:~/lab4$ ./lab4_1.o 10
counter: 0
zmf@zmf:~/lab4$ ./lab4_1.o 100
counter: 0
zmf@zmf:~/lab4$ ./lab4_1.o 1000
counter: 0
zmf@zmf:~/lab4$ ./lab4_1.o 10000
counter: 0
zmf@zmf:~/lab4$ ./lab4_1.o 100000
counter: 100000
zmf@zmf:~/lab4$ ./lab4_1.o 1000000
counter: -715106
zmf@zmf:~/lab4$ ./lab4_1.o 10000000
counter: 1347531
zmf@zmf:~/lab4$ ./lab4_1.o 100000000
counter: -40050340
```

b.

with non atomic instructions

n	Counter
10	0
100	0
100,0	0
100,00	0
100,000	100000
100,000,0	-715106
100,000,00	1347531
100,000,000	-40050340

c.

Answer: It is not a constant.

Reason:

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

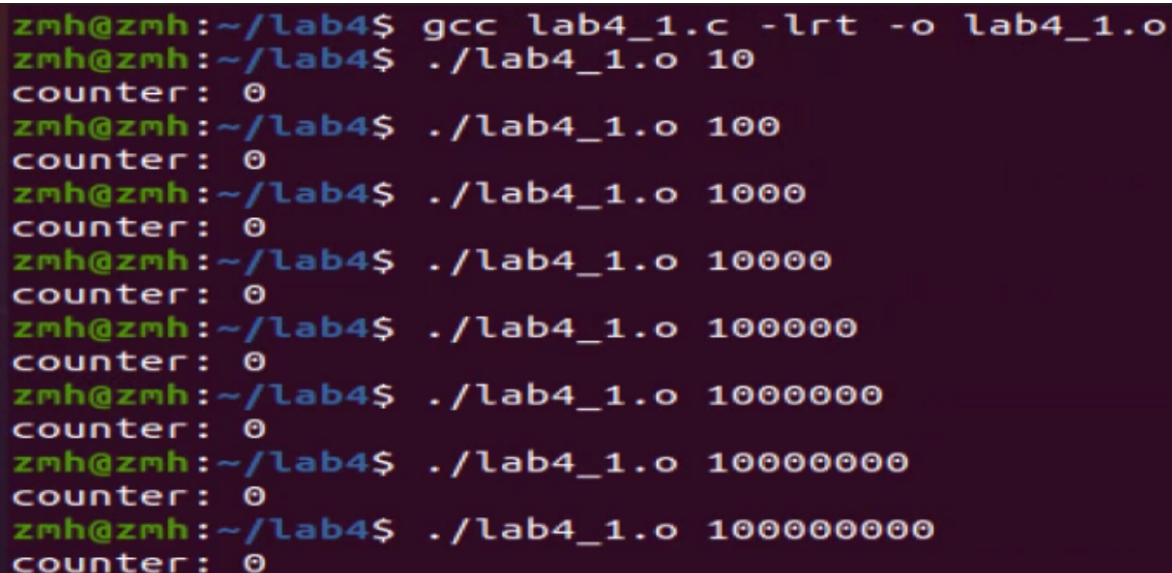
```
*counter = (*counter) + 1; //none atomic
*counter = (*counter) - 1; //none atomic
```

this is a non-atomic instruction which may lead the final value to be non-zero. For example, when parent process is executing add operation, the child process is executing the sub operation at same time. They all change the same variable at the same time. This will lead result 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);
```

the result:



```
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
```

Since they are atomic instructions, the add operation and store operation will be executed just like one instruction which will not be interrupt by another process.

d. Yes

From experiment, we can see that

from 10 ~ 10000, the result will be 0

Start from 100000 to 100000000 and even more larger times, the result will be non-zero.

Reason:

I think may be when a process spend too long time on the job, it will cause time interrupt, and cpu need to switch to execute another process. So during this context switch, it may lead two process to change the variable at the same time. So the final result will be non-zero.

problem 2

a.

```
zmh@zmf:~/lab4$ gcc lab4_2.c -lrt -o lab4_2.o
zmh@zmf:~/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@zmf:~/lab4$
```

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

$5 * \text{sizeof}(\text{int}) = 20\text{B}$, my shared buffer can actually hold 20 bytes of elements. because when the consumer tried to consume the items, it just check whether the item is equal to FLAG or not. If all the 20 bytes items are not equal to FLAG. My buffer can hold 20B elements.