Lab task 2

Q. Consider the functions functionFirst() and functionSecond() defined as follows:

Shared data:

Semaphore s1=1,s2=1,d=1;
int c1=0,c2=0;

```
void functionFirst()
{
        wait(s1);
        c1 = c1 + 1;
        if (c1==1) wait(d);
        signal(s1);

      | Critical Section code |

        wait(s1);
        c1 = c1 - 1;
        if (c1==0) signal(d);
        signal(s1);

      | Remainder Section code |
}
```

```
void functionSecond()
{
        wait(s2);
        c2 = c2 + 1;
        if (c2==1) wait(d);
        signal(s2);

      | Critical Section code |

        wait(s2);
        c2 = c2 - 1;
        if (c2==0) signal(d);
        signal(s2);

      | Remainder Section code |
```

Two different functions involving critical section codes are given above. Assume that there are **N processes** running in our system which may call functionFirst() or functionSecond() including the semaphore operations **wait()** and **signal()**.

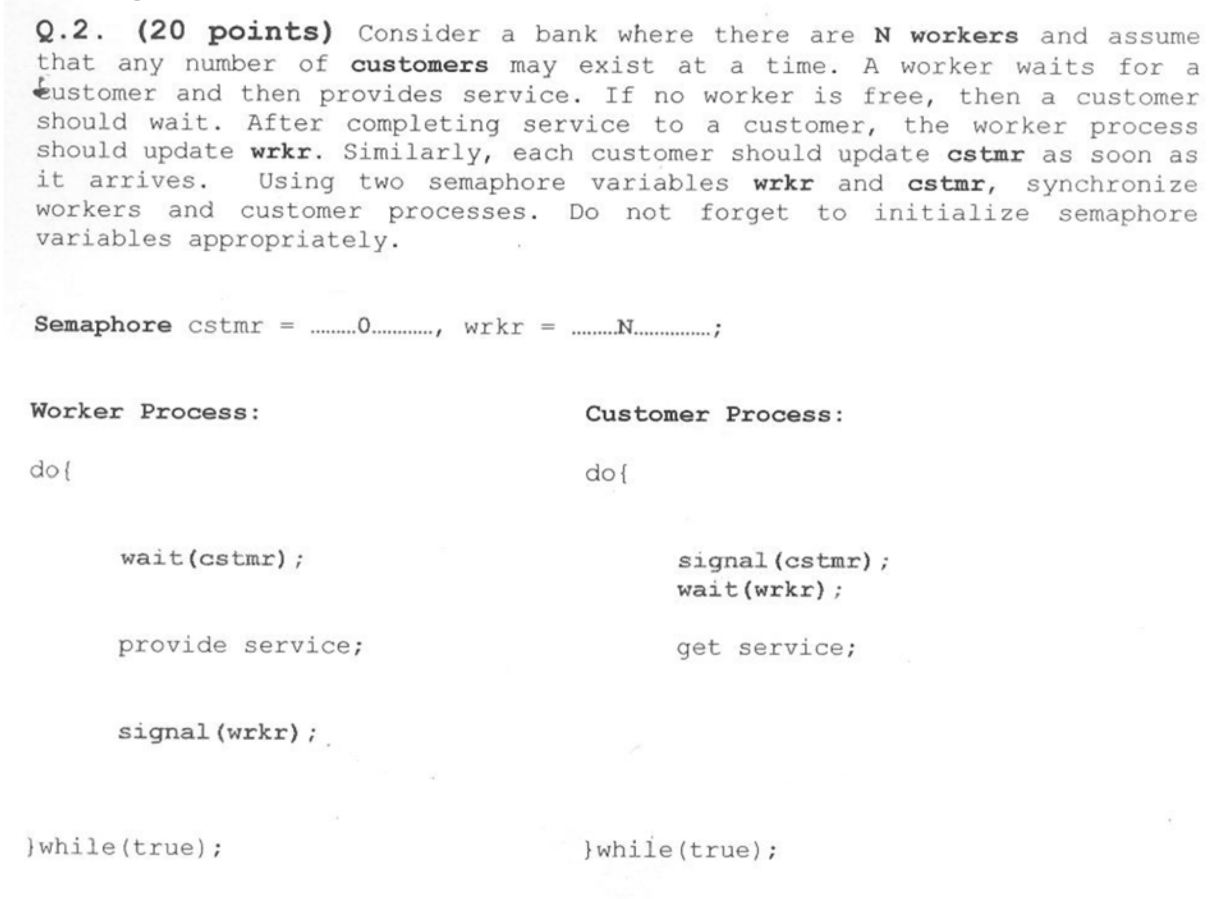
a. How many processes may have functionFirst() calls running in their critical sections? What would be the corresponding d value at that time?

Process count: N	d value: 0

b. While one functionFirst() is in its critical section, how many functionSecond() calls can be running in its critical section? What would be the corresponding d value at that time?

Process count: 0	d value: 0

Lab 2 assignment



Q.2. (20 points) Consider a bank where there are **N workers** and assume that any number of **customers** may exist at a time. A worker waits for a customer and then provides service. If no worker is free, then a customer should wait. After completing service to a customer, the worker process should update **wrkr**. Similarly, each customer should update **cstmr** as soon as it arrives. Using two semaphore variables **wrkr** and **cstmr,** synchronize workers and customer processes. Do not forget to initialize semaphore variables appropriately.

Semaphore cstmr = ...0............, wrkr = ...N...............;

Worker Process:

```
do{

      wait(cstmr);

      provide service;

      signal(wrkr);

}while(true);
```

Customer Process:

```
do{

      signal(cstmr);
      wait(wrkr);

      get service;

}while(true);
```

Q. Consider the **too-much-milk** problem which may occur when 2 persons are sharing the same house.

10

	Person A	**Person B**
3:00	Look in fridge. NO milk.	
3:05	Leave for market.	
3:10	Arrive at market.	Look in fridge. Out of milk.
3:15	Leave market.	Leave for market.
3:20	Arrive home, put milk away.	Arrive at market.
3:25		Leave market.
3:30		Arrive home. Too much milk.

In other words, 2 persons who check the fridge for milk at different times notice that no milk is left. Each goes to the market and buys milk. At the end, there is more milk than necessary.

In order to avoid this, they should be synchronized using semaphores. Assume that the shared data are defined as follows:

Shared data:

```
        Semaphore OKToBuyMilk = 1;
        int NoMilk;
```

Nomilk is true when there is no milk in the fridge. **OKToBuyMilk** is used to achieve mutual exclusion between processes A and B. Before buying milk by calling the function "**BuyMilk**", each process should check whether the value of **OKToBuyMilk** is 1. Otherwise, then should wait. An appropriate function call should be used for this purpose which firstly verifies that **OKToBuyMilk** = 1 and then reduces its value and hence blocks other process. After this is done, the process is in its critical section where the value of **NoMilk** is firstly checked. If there is no milk, then **BuyMilk** is called. After completion, another appropriate function should be called for ending the critical section.

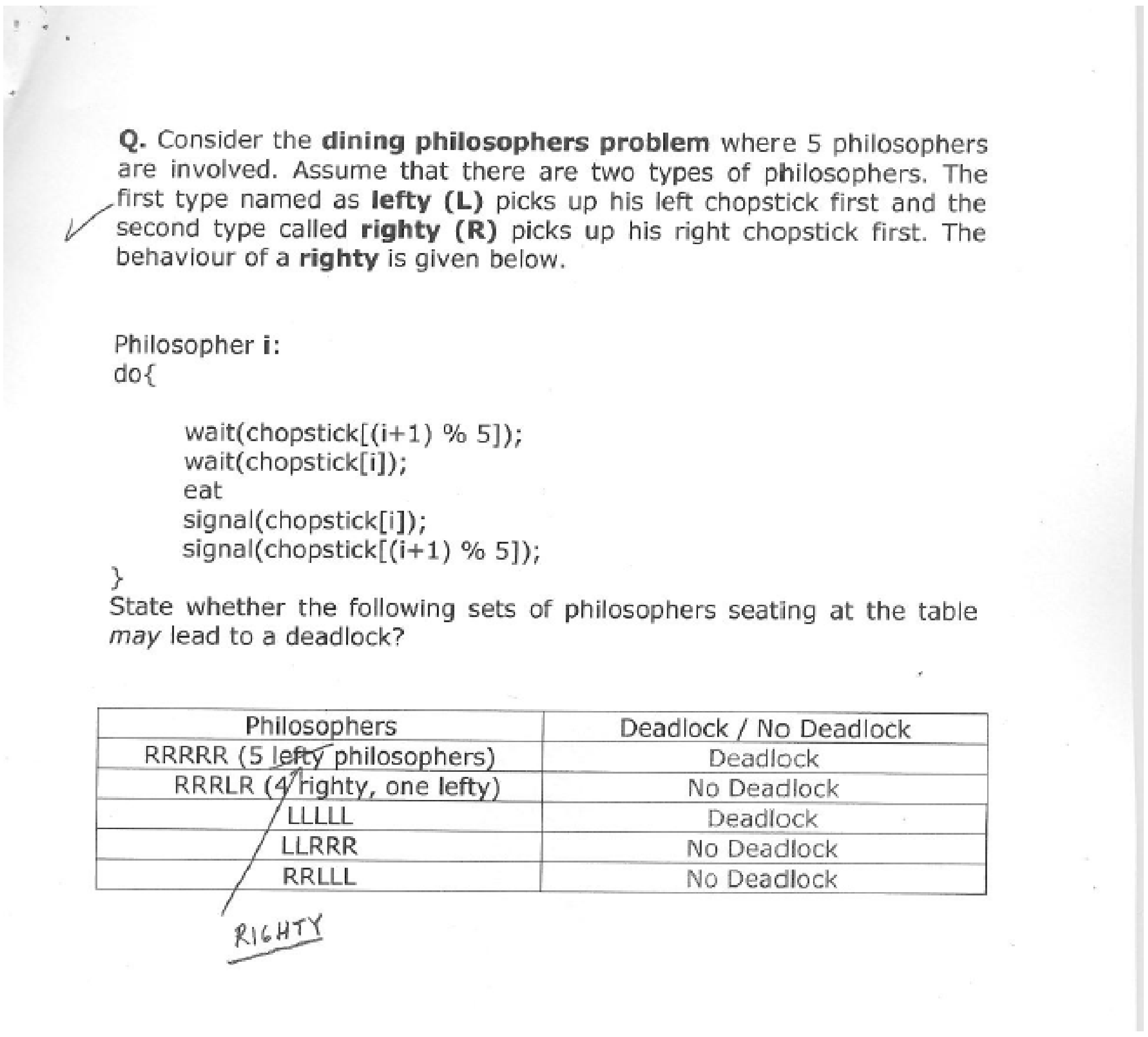
Taking into account this information, please provide the pseudocode for processes A and B.

Processes A and B

```
    wait(OKToBuyMilk);

    if (NoMilk)
         BuyMilk;

    signal(OKToBuyMilk);
```

BuyMilk modifies NoMilk

Lab 3 task



Q. Consider the **dining philosophers problem** where 5 philosophers are involved. Assume that there are two types of philosophers. The first type named as **lefty (L)** picks up his left chopstick first and the second type called **righty (R)** picks up his right chopstick first. The behaviour of a **righty** is given below.

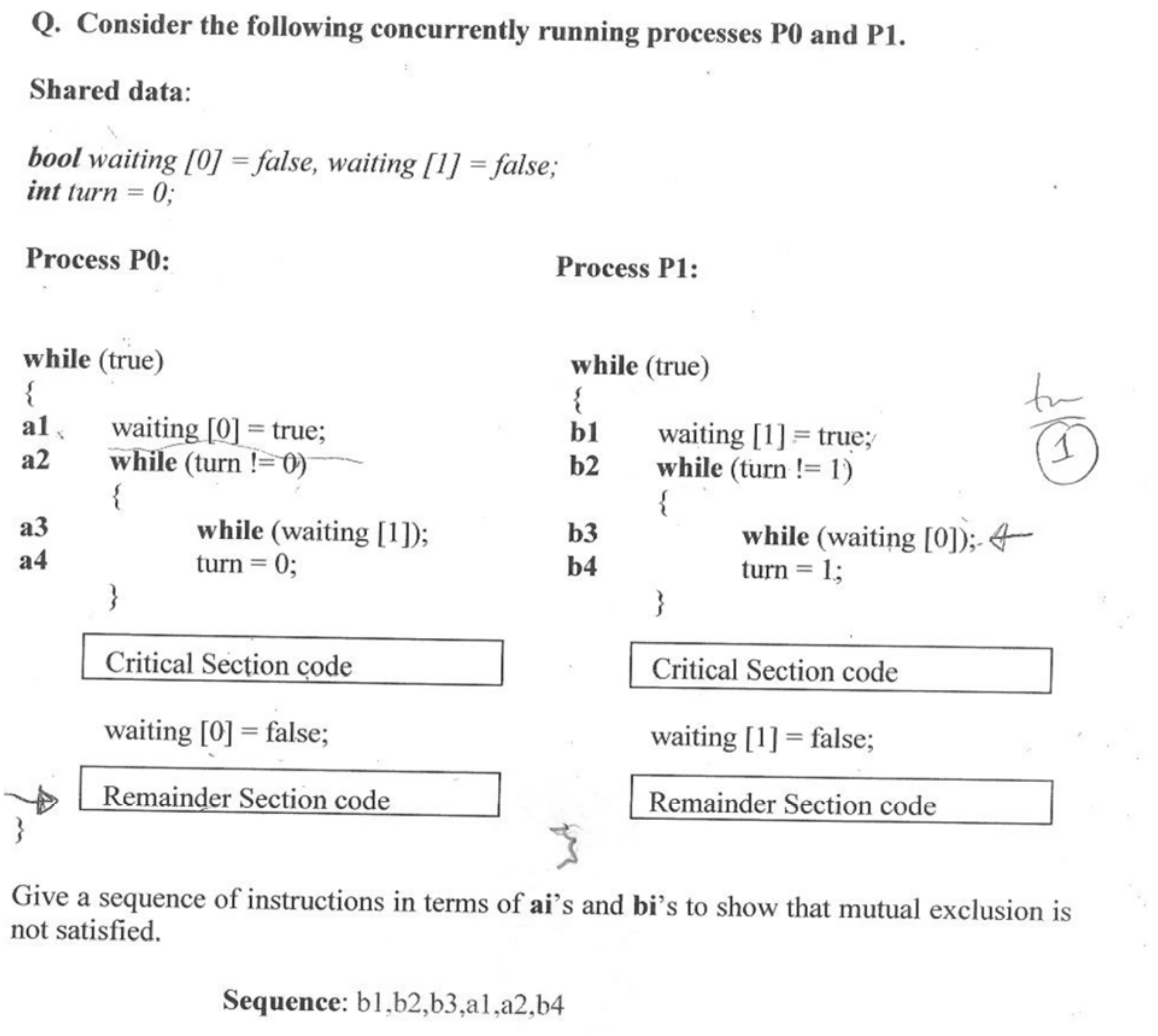
```
Philosopher i:
do{

      wait(chopstick[(i+1) % 5]);
      wait(chopstick[i]);
      eat
      signal(chopstick[i]);
      signal(chopstick[(i+1) % 5]);
}
```

State whether the following sets of philosophers seating at the table *may* lead to a deadlock?

Philosophers	Deadlock / No Deadlock
RRRRR (5 ~~lefty~~ philosophers)	Deadlock
RRRLR (4 righty, one lefty)	No Deadlock
LLLLL	Deadlock
LLRRR	No Deadlock
RRLLL	No Deadlock

RIGHTY

Lab task 3



Q. Consider the following concurrently running processes P0 and P1.

Shared data:

bool *waiting [0] = false, waiting [1] = false;*
int *turn = 0;*

Process P0:

```
while (true)
{
a1      waiting [0] = true;
a2      while (turn != 0)
        {
a3              while (waiting [1]);
a4              turn = 0;
        }
      Critical Section code
        waiting [0] = false;
      Remainder Section code
}
```

Process P1:

```
while (true)
{
b1      waiting [1] = true;
b2      while (turn != 1)
        {
b3              while (waiting [0]);
b4              turn = 1;
        }
      Critical Section code
        waiting [1] = false;
      Remainder Section code
}
```

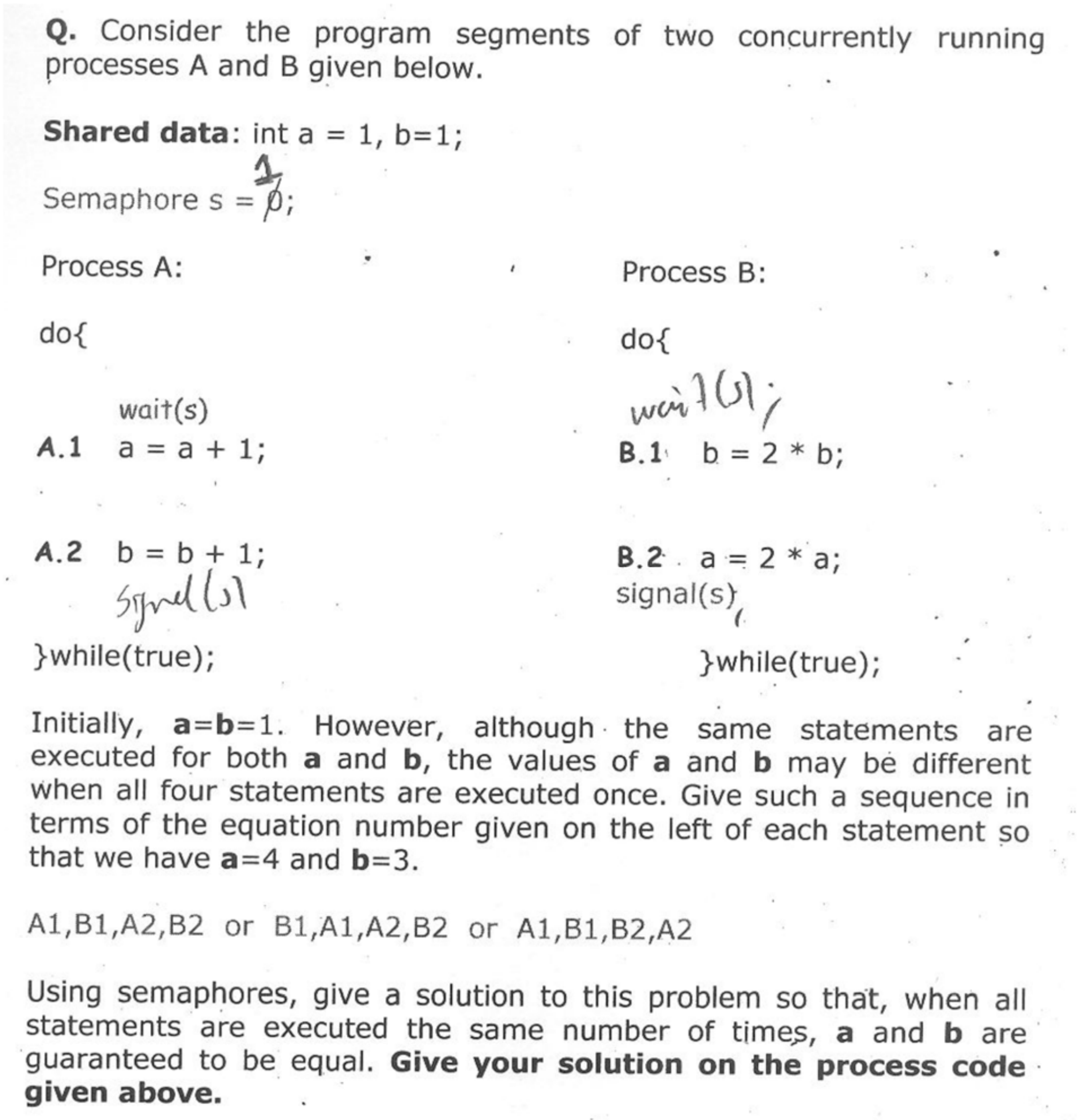
tu

1

Give a sequence of instructions in terms of **ai**'s and **bi**'s to show that mutual exclusion is not satisfied.

Sequence: b1,b2,b3,a1,a2,b4

Lab task 3



Q. Consider the program segments of two concurrently running processes A and B given below.

Shared data: int a = 1, b=1;

Semaphore s = ~~0~~ 1;

```
Process A:

do{

     wait(s)
A.1  a = a + 1;

A.2  b = b + 1;
     signal(s)

}while(true);
```

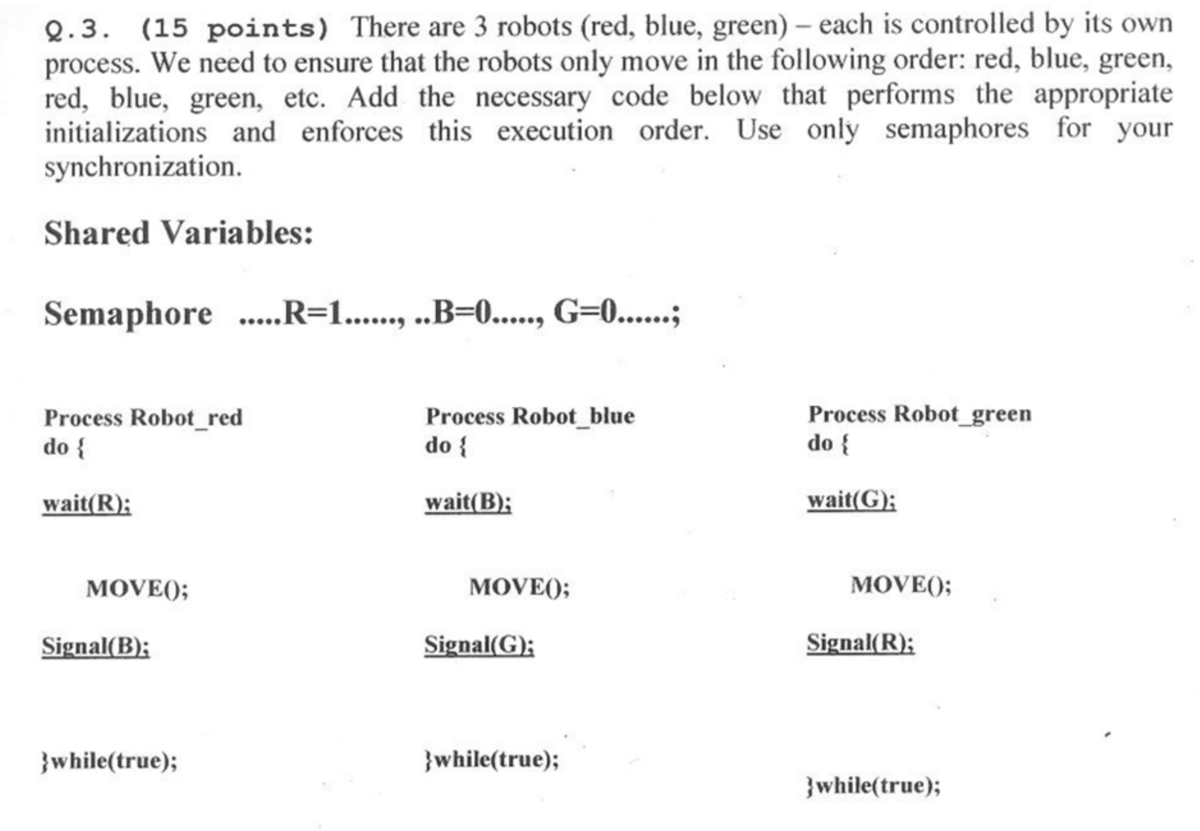
```
Process B:

do{
wait(s);
B.1  b = 2 * b;

B.2  a = 2 * a;
signal(s)

     }while(true);
```

Initially, **a**=**b**=1. However, although the same statements are executed for both **a** and **b**, the values of **a** and **b** may be different when all four statements are executed once. Give such a sequence in terms of the equation number given on the left of each statement so that we have **a**=4 and **b**=3.

A1,B1,A2,B2 or B1,A1,A2,B2 or A1,B1,B2,A2

Using semaphores, give a solution to this problem so that, when all statements are executed the same number of times, **a** and **b** are guaranteed to be equal. **Give your solution on the process code given above.**

Lab task 3



Q.3. (15 points) There are 3 robots (red, blue, green) – each is controlled by its own process. We need to ensure that the robots only move in the following order: red, blue, green, red, blue, green, etc. Add the necessary code below that performs the appropriate initializations and enforces this execution order. Use only semaphores for your synchronization.

Shared Variables:

Semaphore .....R=1......, ..B=0....., G=0......;

```
Process Robot_red        Process Robot_blue       Process Robot_green
do {                     do {                     do {

wait(R);                 wait(B);                 wait(G);


    MOVE();                  MOVE();                  MOVE();

Signal(B);               Signal(G);               Signal(R);



}while(true);            }while(true);
                                                  }while(true);
```