Week 1- SYNC Assignment

Group 3

Table of contents

First Assignment

Problem description

6.1.1 A Critical Section

There are two threads, with the code listed below. The programmer did his best to achieve mutual exclusion, fairness, and to avoid deadlock. However, *mutual exclusion* is not guaranteed.

Answer the following questions:

a) how can both threads enter the Critical Section at the same time?
b) can *deadlock* occur? (why/why not?)
c) is this implementation *fair* (i.e. is it *starvation*-free)?

For all three questions: give a precise description how it happens. Make a table where you write on each line the executed statement (like A1 or B7), together with the actual value of the variables (flag[0], flag[1], lock[0], lock[1]) after execution of that statement.

```
bool flag[2] = { false, false };
bool lock[2] = { false, false };
```

thread A:

```
    while (true)
    {
1.    flag[0] = true;
2.    lock[1] = false;
3.    if (flag[1] == true)
      {
4.      lock[1] = true;
5.      flag[0] = false;
      }
6.    while (lock[1] || flag[1])
      {
7.      flag[0] = false;
8.      flag[0] = true;
      }

9.    CriticalSection();

10.   flag[0] = false;
11.   lock[0] = false;
    }
```

thread B:

```
while (true)
{
  flag[1] = true;
  lock[0] = false;
  if (flag[0] == true)
  {
    lock[0] = true;
    flag[1] = false;
  }
  while (lock[0] || flag[0])
  {
    flag[1] = false;
    flag[1] = true;
  }

  CriticalSection();

  flag[1] = false;
  lock[1] = false;
}
```

Solution

1 - How can both threads enter the Critical Section at the same time?

If you follow this sequence

Flag[0]	Flag[1]	Lock[0]	Lock[1]	Line	Last thread
F	F	F	F	0	
T	-	-	-	A1	
-	-	-	-	A2	
-	-	-	-	A3	
-	-	-	-	A6	
-	-	-	-	A9 (Critical)	
-	T	-	-	B1	
-	-	-	-	B2	
-	-	-	-	B3	
-	-	T	-	B4	
-	F	-	-	B5	
-	-	-	-	B6	
F	-	-	-	A10	
-	-	F	-	A11	
-	-	-	-	B9 (Critical)	Last
T	-	-	-	A1	
-	-	-	-	A2	
-	-	-	-	A3	
-	-	-	-	A6	
-	-	-	-	A9 (Critical)	Last

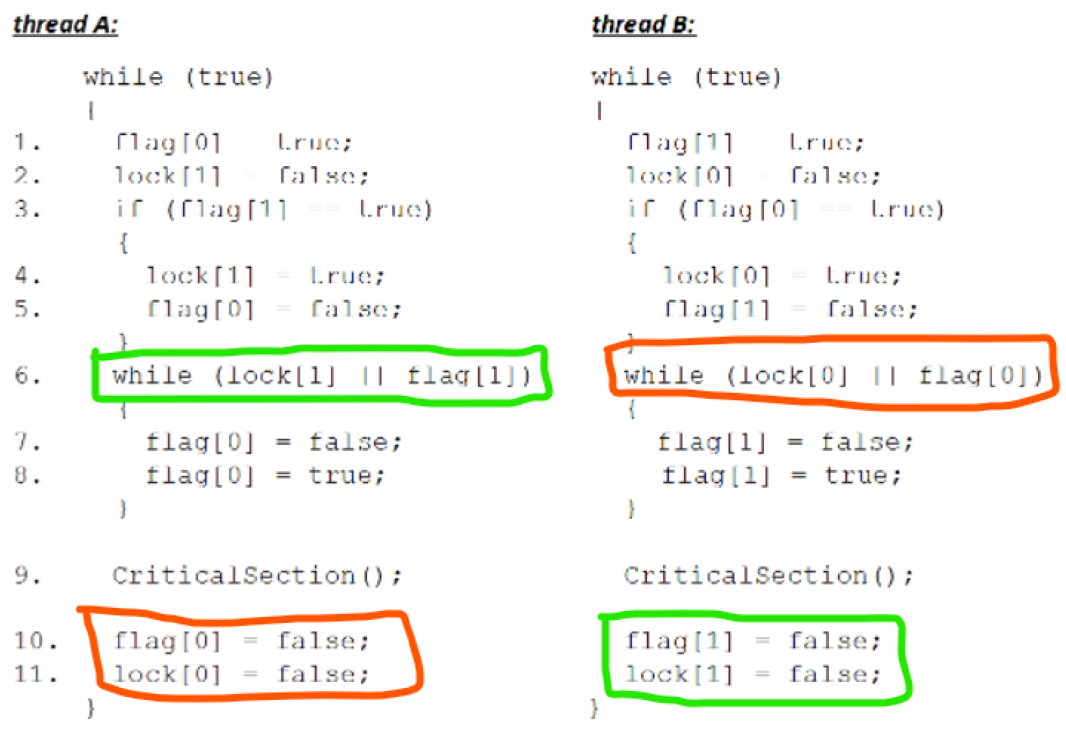
Both the threads will be in the critical section at the same time.

2 - Can deadlock occur? (why/why not?)

Flag[0]	Flag[1]	Lock[0]	Lock[1]	Line	Last thread
F	F	F	F	0	
T	-	-	-	A1	
-	T	-	-	B1	
-	-	-	-	A2	
-	-	-	-	A3	
-	-	-	T	A4	
-	-	-	-	B2	
-	-	-	-	B3	
-	-	T	-	B4	
F	-	-	-	A5	
-	-	-	-	A6	Stuck in loop
-	F	-	-	B5	
-	-	-	-	B6	Stuck in loop

Both thread 0 and thread 1 are stuck in a while loop, just turning each other's flags on and off. Neither will reach the critical section after that, because lock[0] and lock[1] are both true. Therefore a deadlock occurs.

3 - Is this implementation fair (i.e. is it starvation-free)?



thread A:

```
     while (true)
     {
1.     flag[0] = true;
2.     lock[1] = false;
3.     if (flag[1] == true)
       {
4.       lock[1] = true;
5.       flag[0] = false;
       }
6.     while (lock[1] || flag[1])
       {
7.       flag[0] = false;
8.       flag[0] = true;
       }

9.     CriticalSection();

10.    flag[0] = false;
11.    lock[0] = false;
     }
```

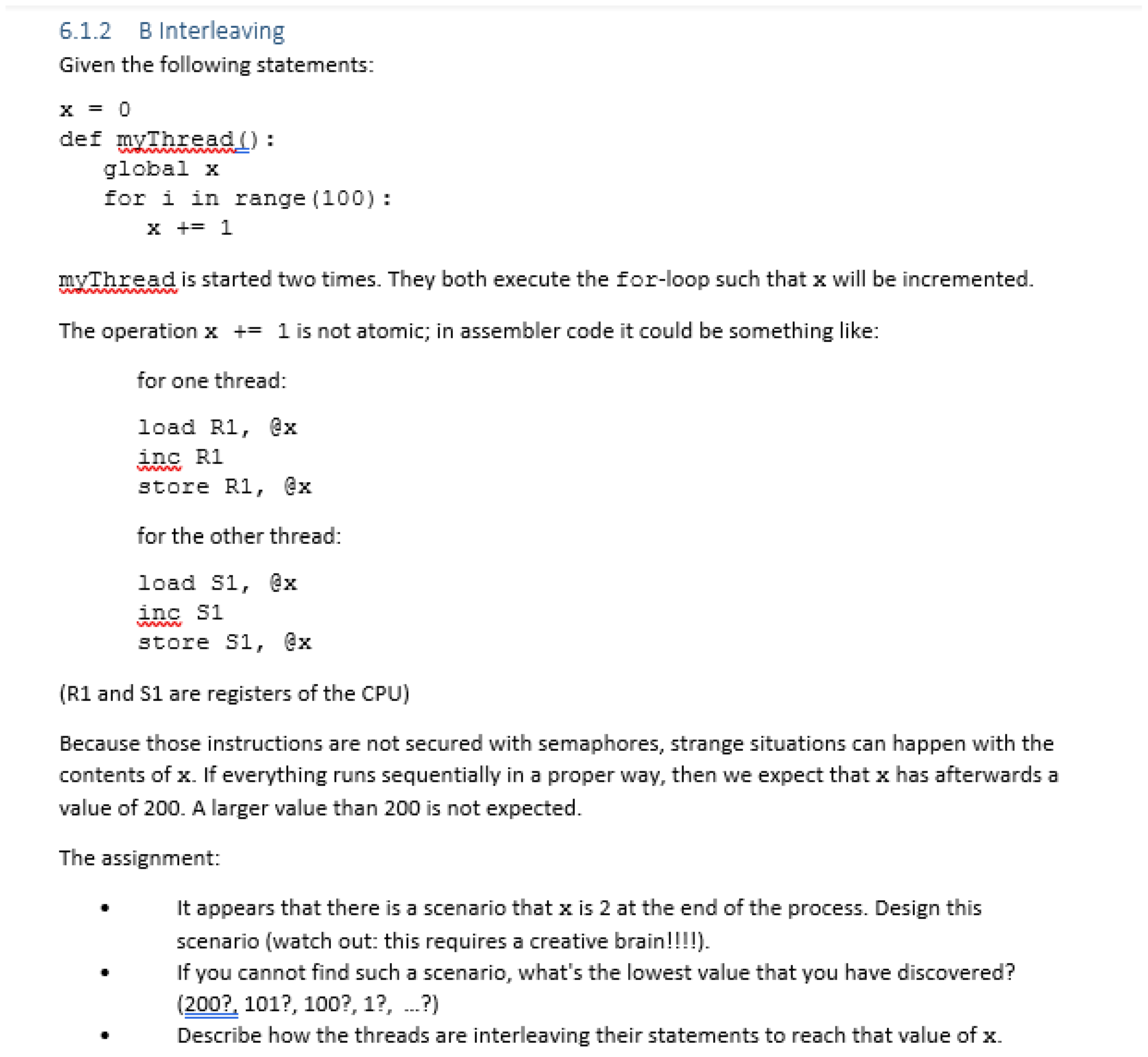
thread B:

```
while (true)
{
  flag[1] = true;
  lock[0] = false;
  if (flag[0] == true)
  {
    lock[0] = true;
    flag[1] = false;
  }
  while (lock[0] || flag[0])
  {
    flag[1] = false;
    flag[1] = true;
  }

  CriticalSection();

  flag[1] = false;
  lock[1] = false;
}
```

We believe that this algorithm is starvation-free, because if Thread 0 is executed first until line 11, then flag[0] and lock[0] will be set to false, resulting in Thread 1 not being able to perform the while loop inside it. Same logic also applies if Thread 1 runs first.

Second Assignment

Problem description



6.1.2 B Interleaving

Given the following statements:

```
x = 0
def myThread():
   global x
   for i in range(100):
      x += 1
```

`myThread` is started two times. They both execute the `for`-loop such that `x` will be incremented.

The operation `x += 1` is not atomic; in assembler code it could be something like:

for one thread:

```
load R1, @x
inc R1
store R1, @x
```

for the other thread:

```
load S1, @x
inc S1
store S1, @x
```

(R1 and S1 are registers of the CPU)

Because those instructions are not secured with semaphores, strange situations can happen with the contents of `x`. If everything runs sequentially in a proper way, then we expect that `x` has afterwards a value of 200. A larger value than 200 is not expected.

The assignment:

- It appears that there is a scenario that `x` is 2 at the end of the process. Design this scenario (watch out: this requires a creative brain!!!!).
- If you cannot find such a scenario, what's the lowest value that you have discovered? (200?, 101?, 100?, 1?, …?)
- Describe how the threads are interleaving their statements to reach that value of `x`.

Solution

Let’s consider the following way of executing the two threads as:

Thread 1

```
load R1, @x
inc R1
store R1, @x
```

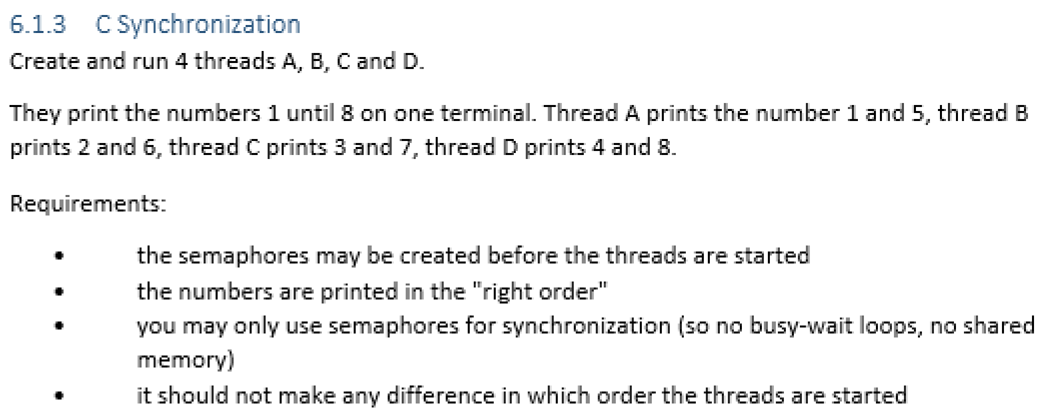
Thread 2

```
load S1, @x
inc S1
store S1, @x
```

First, both threads will copy the value of x into their registers. We start with Thread 1 which performs the first 99 iterations, setting x = 99. This value will be stored into the cache, but the value of x in Thread 2 is still 0.
Thread 2 will perform the first iteration and will increment x (x = 1). Now the value of x is overwritten also in Thread 1 which is stored inside its register.
Thread 2 will now perform the remaining 99 iterations setting x = 100, but since Thread 1 already has x = 1, it will get incremented reaching 2. The value is finally overwritten also in Thread 2.
Therefore, we can conclude that the maximum value x can get is 100 and the minimum value is 2.

Third assignment

Problem description



6.1.3 C Synchronization

Create and run 4 threads A, B, C and D.

They print the numbers 1 until 8 on one terminal. Thread A prints the number 1 and 5, thread B prints 2 and 6, thread C prints 3 and 7, thread D prints 4 and 8.

Requirements:

- the semaphores may be created before the threads are started
- the numbers are printed in the "right order"
- you may only use semaphores for synchronization (so no busy-wait loops, no shared memory)
- it should not make any difference in which order the threads are started

Solution

To solve this assignment, we have created 4 semaphores and 4 threads, as mentioned. In order to make sure that the code will run in the right order we initially set the value of semaphore A to 1 and the other semaphores to 0 to wait for semaphore A to send a signal.

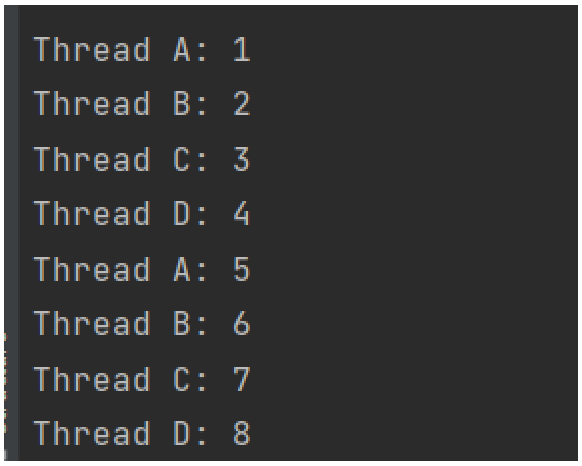
```
semaphoreA = MySemaphore(1, "semaphoreA")
semaphoreB = MySemaphore(0, "semaphoreB")
semaphoreC = MySemaphore(0, "semaphoreC")
semaphoreD = MySemaphore(0, "semaphoreD")
```

We make sure that each thread will print the numbers which are mentioned in the assignment. (Thread A: 1 and 5) (Thread B: 2 and 6) (Thread C: 3 and 7) (Thread D: 4 and 8)

```
def threadA():
   while True:
       semaphoreA.wait()
       print(1)
       semaphoreB.signal()
       semaphoreA.wait()
       print(5)
       semaphoreB.signal()

def threadB():
   while True:
       semaphoreB.wait()
       print(2)
       semaphoreC.signal()
```

```
        semaphoreB.wait()
        print(6)
        semaphoreC.signal()

def threadC():
    while True:
        semaphoreC.wait()
        print(3)
        semaphoreD.signal()
        semaphoreC.wait()
        print(7)
        semaphoreD.signal()

def threadD():
    while True:
        semaphoreD.wait()
        print(4)
        semaphoreA.signal()
        semaphoreD.wait()
        print(8)
        semaphoreA.signal()
```

Output

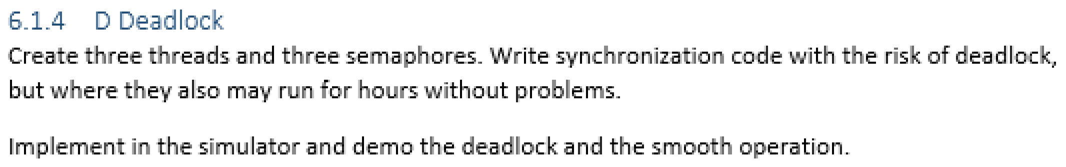


```
Thread A: 1
Thread B: 2
Thread C: 3
Thread D: 4
Thread A: 5
Thread B: 6
Thread C: 7
Thread D: 8
```

A video of this assignment is included alongside this document.

Fourth assignment

Problem description



6.1.4 D Deadlock

Create three threads and three semaphores. Write synchronization code with the risk of deadlock, but where they also may run for hours without problems.

Implement in the simulator and demo the deadlock and the smooth operation.

Solution

To write synchronization code with the risk of deadlock, but where they also may run for hours without problems. We have created three threads with 3 semaphores. The code as follows:

```
def setup():
   subscribe_thread(threadA)
   subscribe_thread(threadB)
   subscribe_thread(threadC)
```

```
def threadA():
  while True:
      (a1) semA.wait()
      (a2) semC.wait()
      (a3) semC.signal()
      (a4) semA.signal()
```

```
def threadB():
  while True:
      (b1)  semB.wait()
      (b2)  semA.wait()
      (b3)  semA.signal()
      (b4)  semB.signal()
```

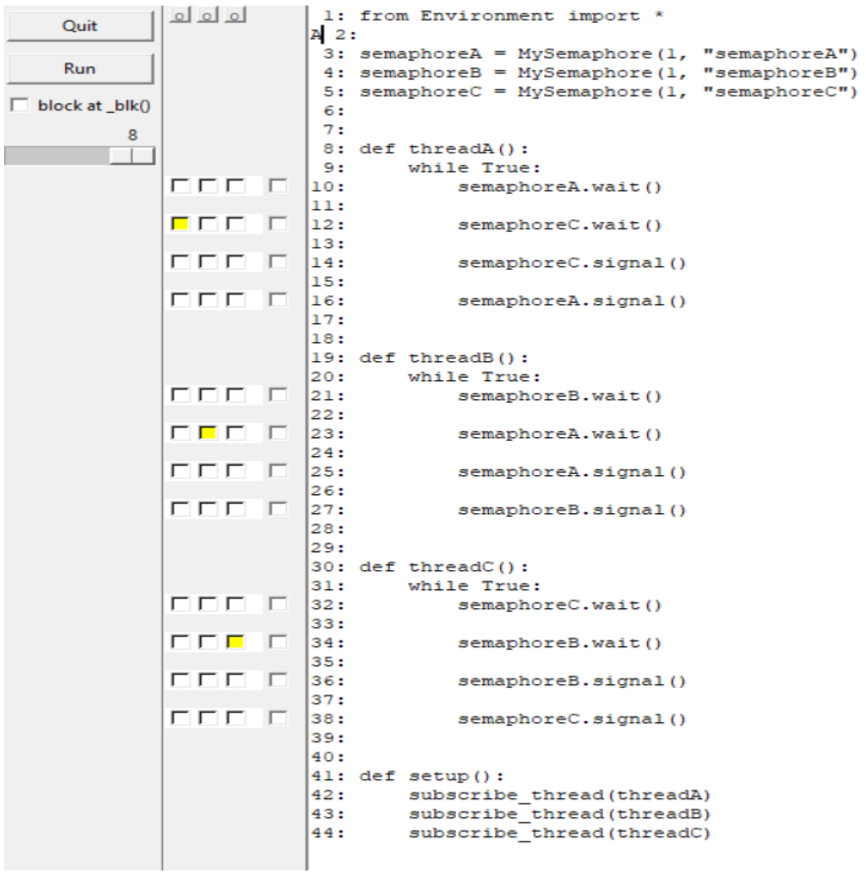
```
def threadC():
  while True:
      (c1) semC.wait()
      (c2) semB.wait()
      (c3) semB.signal()
      (c4) semC.signal()
```

Solution: A1 => A2 => A3 => A4 => B1 => B2 => B3 => B4 => C1 => C2 => C3 => C4. In this order the code will run for hours without problems.

Deadlock: A1 => B1 => C1 => A2 => B2 => C2. Then all semaphores will get stuck and will wait for each other indefinitely.

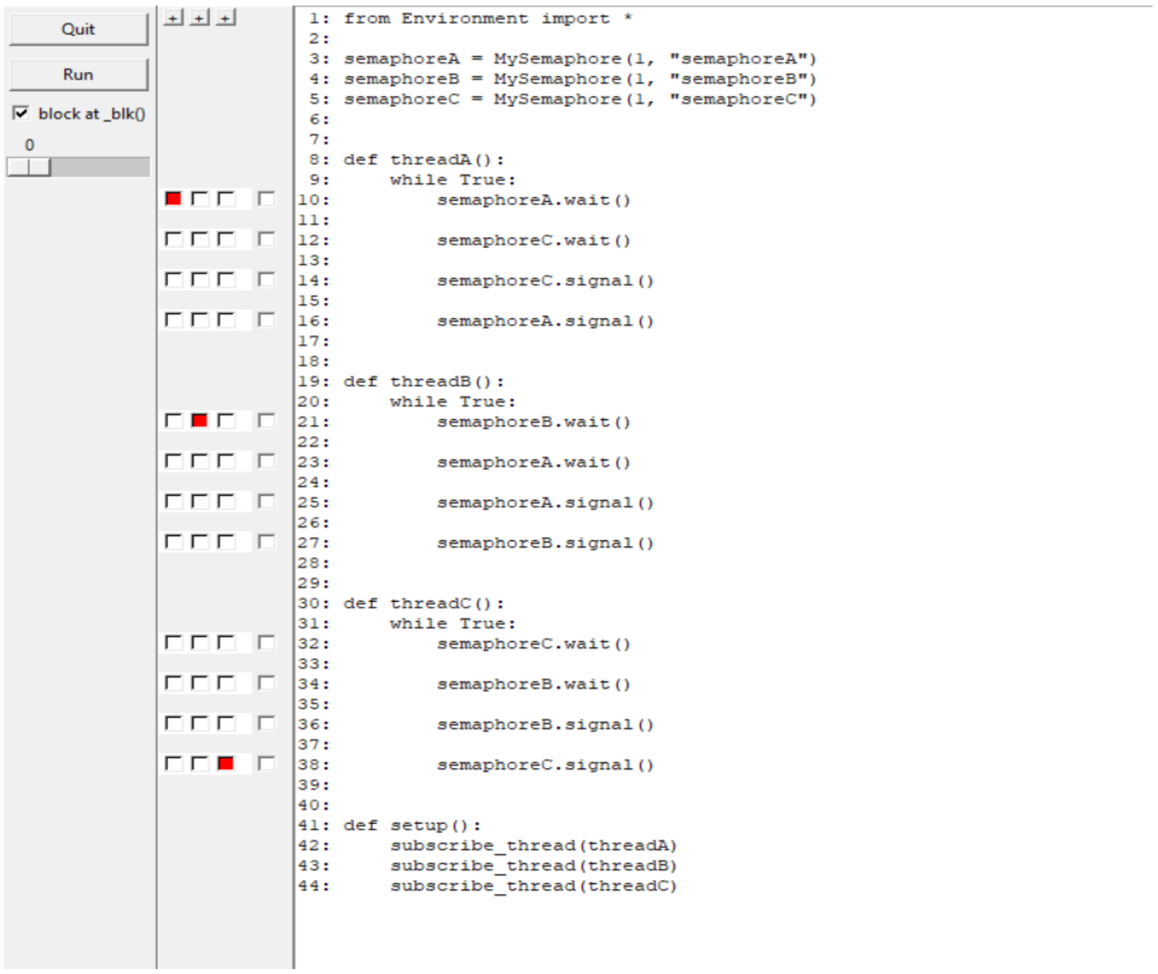
Output

Deadlock state



```
from Environment import *

semaphoreA = MySemaphore(1, "semaphoreA")
semaphoreB = MySemaphore(1, "semaphoreB")
semaphoreC = MySemaphore(1, "semaphoreC")


def threadA():
    while True:
        semaphoreA.wait()

        semaphoreC.wait()

        semaphoreC.signal()

        semaphoreA.signal()


def threadB():
    while True:
        semaphoreB.wait()

        semaphoreA.wait()

        semaphoreA.signal()

        semaphoreB.signal()


def threadC():
    while True:
        semaphoreC.wait()

        semaphoreB.wait()

        semaphoreB.signal()

        semaphoreC.signal()


def setup():
    subscribe_thread(threadA)
    subscribe_thread(threadB)
    subscribe_thread(threadC)
```

Regular Operations



```
from Environment import *

semaphoreA = MySemaphore(1, "semaphoreA")
semaphoreB = MySemaphore(1, "semaphoreB")
semaphoreC = MySemaphore(1, "semaphoreC")


def threadA():
    while True:
        semaphoreA.wait()

        semaphoreC.wait()

        semaphoreC.signal()

        semaphoreA.signal()


def threadB():
    while True:
        semaphoreB.wait()

        semaphoreA.wait()

        semaphoreA.signal()

        semaphoreB.signal()


def threadC():
    while True:
        semaphoreC.wait()

        semaphoreB.wait()

        semaphoreB.signal()

        semaphoreC.signal()


def setup():
    subscribe_thread(threadA)
    subscribe_thread(threadB)
    subscribe_thread(threadC)
```

Videos are provided for this assignment alongside this document.