## Week 1

### A Critical Section

There are two threads, with the code listed below.

*Mutual exclusion* is not guaranteed. Prove this by giving a sequence of statements, such that both processes enter their Critical Section at the same time.

**Answer:**

If we follow this sequence:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| flag[0] | flag[1] | lock[0] | lock[1] | thred 0 | thread 1 |
| FALSE | FALSE | FALSE | FALSE | 0 | 0 |
| FALSE | TRUE | FALSE | FALSE | 0 | 15 |
| FALSE | TRUE | FALSE | TRUE | 9 | 15 |
| FALSE | FALSE | FALSE | FALSE | 9 | 0 |
| FALSE | FALSE | FALSE | FALSE | 15 | 0 |
| FALSE | TRUE | FALSE | FALSE | 15 | 15 |

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

Answer the following questions as well:

* can *deadlock* occur? (why/why not?)

**Answer:**

If we follow this sequence:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| flag[0] | flag[1] | lock[0] | lock[1] | thread 0 | thread 1 |
| FALSE | FALSE | FALSE | FALSE | 0 | 0 |
| TRUE | FALSE | FALSE | FALSE | 2 | 0 |
| TRUE | TRUE | FALSE | FALSE | 2 | 2 |
| TRUE | TRUE | FALSE | TRUE | 6 | 2 |
| TRUE | TRUE | TRUE | TRUE | 6 | 6 |
| TRUE | TRUE | TRUE | TRUE | 9 | 6 |
| TRUE | TRUE | TRUE | TRUE | 9 | 9 |

Both thread 0 and thread 1 will get stuck in the while loop only toggling the flags of the other on and off. Neither will ever reach the critical section after as both lock[0] and lock [1] are true. So yes deadlock can occur.

* is this implementation *fair* (i.e. is it *starvation*-free)?  
  **Answer:**

This algorithm is not starvation-free. If one of the threads ends up in the while loop and keeps setting the flag variable to false when the other thread reaches the if-statement it will continuously be stuck inside the while loop, starving the thread.

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

***thread 0: thread 1:***

0: while (true) while (true)

1: { {

2: flag[0] = true; flag[1] = true;

3: lock[1] = false; lock[0] = false;

4: if (flag[1] == true) if (flag[0] == true)

5: { {

6: lock[1] = true; lock[0] = true;

7: flag[0] = false; flag[1] = false;

8: } }

9: while (lock[1] || flag[1]) while (lock[0] || flag[0])

10: { {

11: flag[0] = false; flag[1] = false;

12: flag[0] = true; flag[1] = true;

13: } }

14:

15: CriticalSection(); CriticalSection();

16:

17: flag[0] = false; flag[1] = false;

18: lock[0] = false; lock[1] = false;

19:} }

### 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.

**Answer:**

Let thread 2 get value S1. This is currently 0. Then thread 1 loops from 0 to 98 so the current actual value for x would be 99. Then thread 2 runs once and sets x to 1. The value of S1 that thread 2 had was currently 0 so we increase that by 1. Then thread 1 takes value R1 = 1. We run threads 2 98 more times until the for loop stops. We then run the last iteration for thread 1 which sets x to 2. R1 = 1 add 1 to it.

### C Synchronization

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

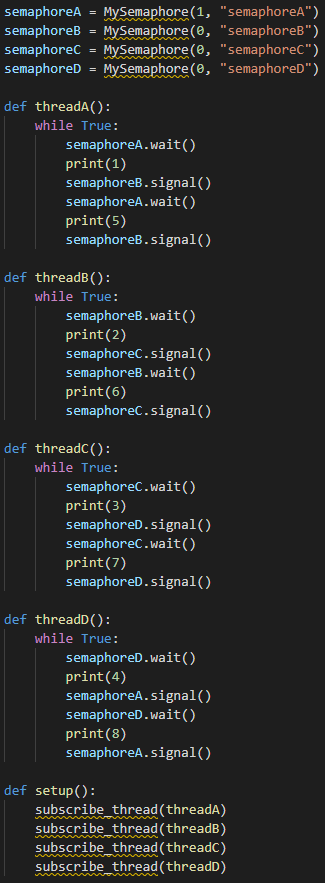
They print the numbers 1 until 8 on one terminal. Thread A prints the number 1 and 5, thread B prints 2 and 6, thread C prints 3 and 7, thread D prints 4 and 8.

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

Answer:

The following code was used to get the desired result:



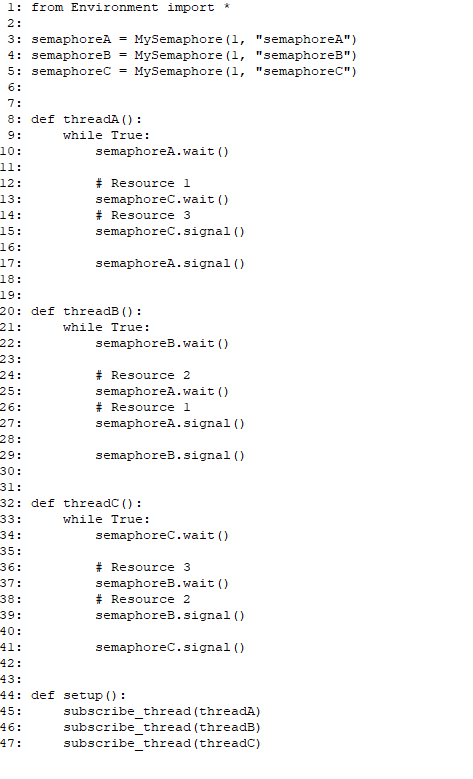
### D Deadlock

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

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

**Answer**:

For this we created this synchronization code:



This code has all the properties for deadlock to occur.

**Mutual exclusion**

This code checks make sure that resources are not used by two threads at the same time. That is why this code has mutual exclusion.

**Hold and Wait**

All of the threads hold a different resource. Thread A by default first uses resource 1 then it wants resource 3. If semaphore C locks thread A it will not first release Resource 1. This is present with different resources in all threads.

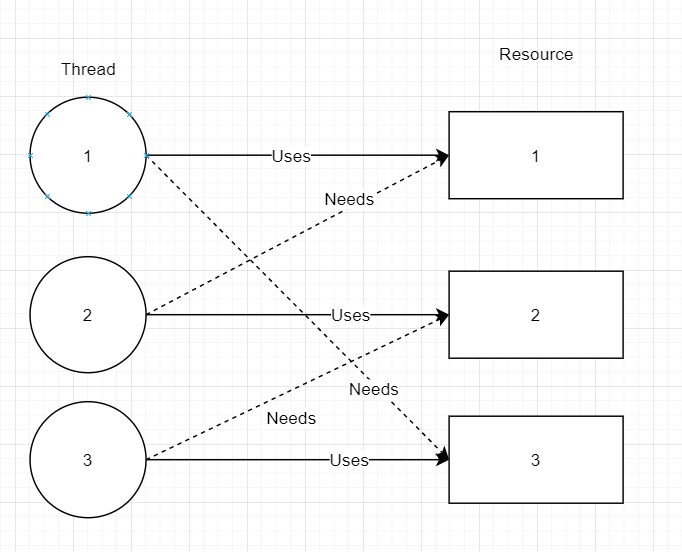
**Circular Waiting**

All threads hold a specific resource at starts. This means that if for example Resource 1 would already be in use. Thread A would have to wait for thread C to finish. So we also have Circular waiting.

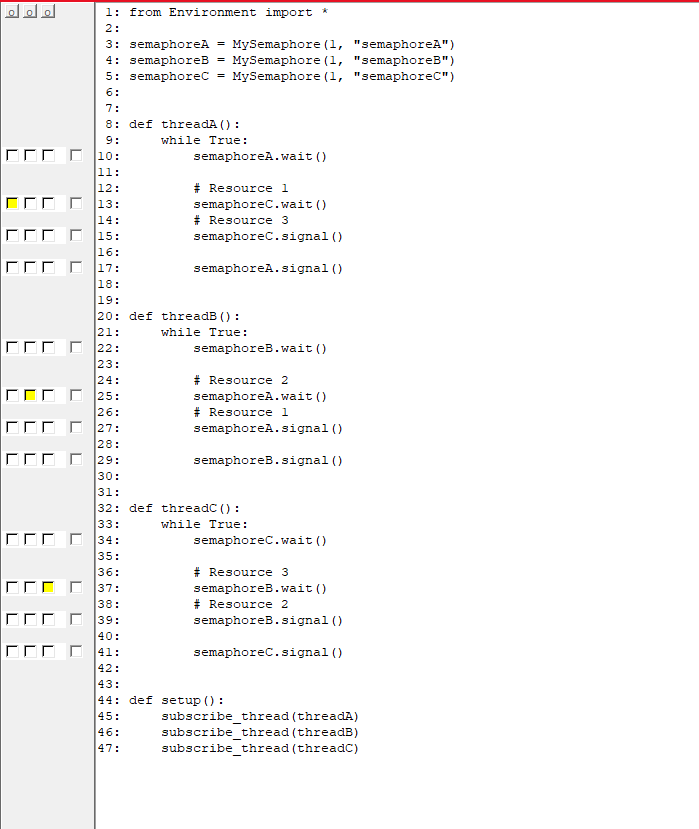
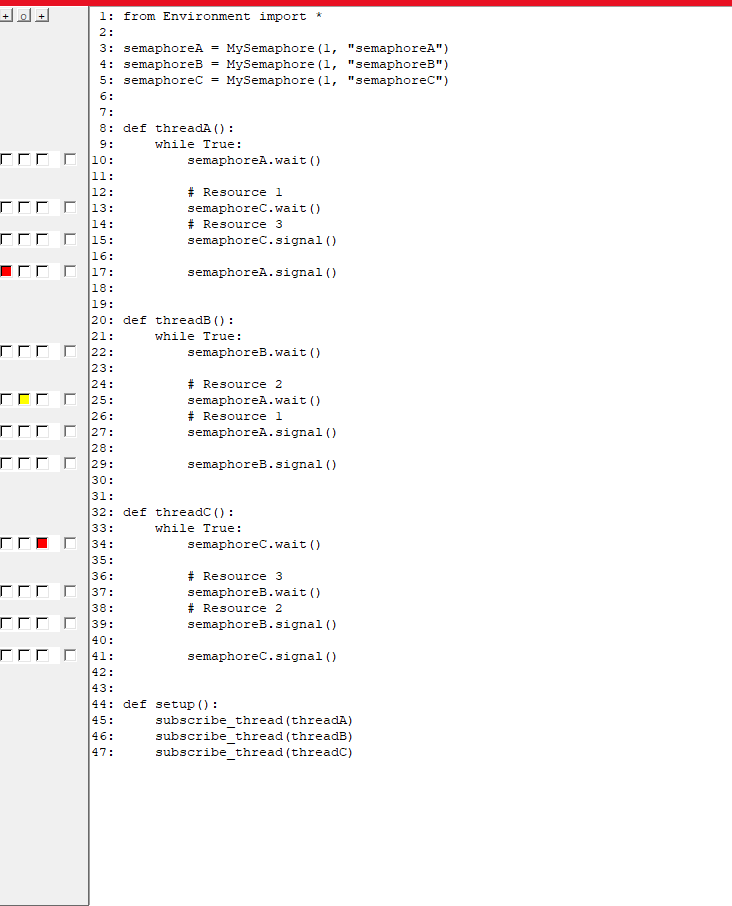
**No preemption**

This is by far the easiest condition to get because you don’t have to do anything. We currently don’t free resources if our we detected a deadlock.

This image illustrates what resource uses to achieve deadlock:

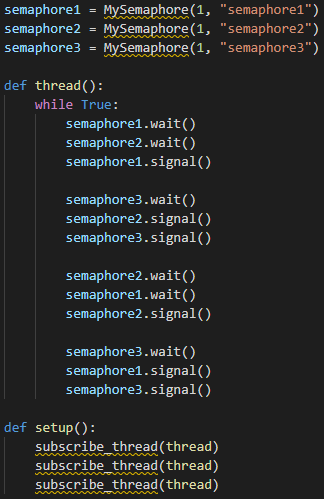


Deadlock state: Regular operations:



**Alternative answer:**

The following code is symmetric, and only uses one function that is run by all 3 threads:



This code runs smoothly if all thread enters and exit one at a time, or if following the execution path described in the table down below.

Additionally, the code can also deadlock under the right circumstances. One such circumstance is described in the table down below.

Execution order for normal operation:

|  |  |  |  |
| --- | --- | --- | --- |
| Thread + line | Semaphore 1 value | Semaphore 2 value | Semaphore 3 value |
| - | 1 | 1 | 1 |
| A: 1 | 0 | 1 | 1 |
| A: 2 | 0 | 0 | 1 |
| B: 1 | -1 | 0 | 1 |
| A: 3 | 0 | 0 | 1 |
| B: 2 | 0 | -1 | 1 |
| C: 1 | -1 | -1 | 1 |
| A: 4 | -1 | -1 | 0 |
| A: 5 | -1 | 0 | 0 |
| B: 3 | 0 | 0 | 0 |
| B: 4 | 0 | 0 | -1 |
| C: 2 | 0 | -1 | -1 |
| A: 6 | 0 | -1 | 0 |
| A: 7 | 0 | -2 | 0 |
| B: 5 | 0 | -1 | 0 |
| C: 3 | 1 | -1 | 0 |
| C: 4 | 1 | -1 | -1 |
| B: 6 | 1 | -1 | 0 |
| C: 5 | 1 | 0 | 0 |
| A: 8 | 0 | 0 | 0 |
| B: 7 | 0 | -1 | 0 |
| C: 6 | 0 | -1 | 1 |
| A: 9 | 0 | 0 | 1 |
| B: 8 | -1 | 0 | 1 |
| C: 7 | -1 | -1 | 1 |
| A: 10 | -1 | -1 | 0 |
| A: 11 | 0 | -1 | 0 |
| B: 9 | 0 | 0 | 0 |
| B: 10 | 0 | 0 | -1 |
| C: 8 | -1 | 0 | -1 |
| A: 12 | -1 | 0 | 0 |
| B: 11 | 0 | 0 | 0 |
| C: 9 | 0 | 1 | 0 |
| C: 10 | 0 | 1 | -1 |
| B: 12 | 0 | 1 | 0 |
| C: 11 | 1 | 1 | 0 |
| C: 12 | 1 | 1 | 1 |

Execution order for deadlock:

|  |  |  |  |
| --- | --- | --- | --- |
| Thread + line | Semaphore 1 value | Semaphore 2 value | Semaphore 3 value |
| - | 1 | 1 | 1 |
| A: 1 | 0 | 1 | 1 |
| A: 2 | 0 | 0 | 1 |
| A: 3 | 1 | 0 | 1 |
| A: 4 | 1 | 0 | 0 |
| A: 5 | 1 | 1 | 0 |
| A: 6 | 1 | 1 | 1 |
| A: 7 | 1 | 0 | 1 |
| B: 1 | 0 | 0 | 1 |
| B: 2 | 0 | -1 | 1 |
| C: 1 | -1 | -1 | 1 |
| A: 8 | -2 | -1 | 1 |