Assignment 3: Semaphores

ETH Zurich

1 Interleaving with Semaphores

1.1 Background

This task is also taken from *Foundations of Multithreaded, Parallel, and Distributed Programming* [1].

1.2 Task

Given the following processes and code, give the possible outputs of the interleavings:

s.count := 0 r.count := 1 x := 0		
P_1	P_2	P_3
s.down	r.down	r.down
r.down	x := x * (x + 1)	x := x + 2
x := x * 2	r.up	r.up
r.up	s.up	

1.3 Solution

The possible execution orders are
P_2, P_1, P_3 or P_2, P_3, P_1 or P_3, P_2, P_1.
The corresponding outputs are: 2, 4, 12.

2 Barriers

2.1 Background

Consider a program with three types of threads: A, B and C. A barrier synchronizes one A-thread and two B-threads, which pass the barrier together as an A-B-B-group. C-threads only wait at the barrier if the number of A-B-B-groups that have already passed the barrier is smaller than or equal to the number of C-threads that have already passed the barrier.

For example, this would be a valid execution:

1. B-thread arrives at the barrier and waits
2. A-thread arrives at the barrier and waits
3. A-thread arrives at the barrier and waits
4. B-thread arrives at the barrier and waits
5. One A-thread and two B-threads pass the barrier
6. C-thread arrives at the barrier and immediately passes
7. B-thread arrives at the barrier and waits

8. C-thread arrives at the barrier and waits
9. B-thread arrives at the barrier and waits
10. One A-thread and two B-threads, as well as one C-thread pass the barrier

2.2 Task

Write synchronization code for threads of type A, B and C that enforces these constraints using semaphores. Your solution may not rely on the number of running threads. You may use Eiffel, Java, or appropriate pseudocode.

You will probably find it helpful to use integer counters for keeping track of the number of waiting threads of type A and B, and a semaphore for each type of thread A, B and C (in addition to further variables and semaphores that might be needed).

Note that you have to initialize all variables and semaphores with the proper values first.

2.3 Solution

```
feature
    a, b: INTEGER
    mutex: SEMAPHORE
    a_queue, b_queue, c_queue: SEMAPHORE

    make
        do
            create mutex.make (1)
            create a_queue.make (0)
            create b_queue.make (0)
            create c_queue.make (0)
        end
```

```
a_thread
    do
        mutex.down
        if (b >= 2) then
            b := b - 2
            mutex.up
            b_queue.up
            b_queue.up
            c_queue.up
        else
            a := a + 1
            mutex.up
            a_queue.down
        end
    end
```

```
b_thread
    do
        mutex.down
        if (b >= 1 and a >= 1) then
            b := b - 1
            a := a - 1
```

```
            mutex.up
            b_queue.up
            a_queue.up
            c_queue.up
        else
            b := b + 1
            mutex.up
            b_queue.down
        end
    end
```

```
c_thread
    do
        c_queue.down
    end
```

3 Unisex bathroom

3.1 Background

This task has been adapted from *Foundations of Multithreaded, Parallel, and Distributed Programming* [1]. In an office there is a unisex bathroom with n toilets. The bathroom is open to both men and women, but it cannot be used by men and women at the same time.

3.2 Task

1. Develop a Java program that simulates the above scenario using semaphores from the Java concurrency library. Your solution should be deadlock free, but it does not have to be starvation free.

2. Justify why your solution is deadlock free.

3.3 Solution

The program and the justifications can be found in the source that comes with this solution.

References

[1] Gregory R. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley, 1999.