# CONCSYS - ASSIGNMENT 4

Upload your solution on ILIAS as a ZIP or TAR.GZ including all your files (sources, text file with explanations, ...). If you have questions, please address them during the lab, on the forum or by email to maria.carpen-amarie@unine.ch.

**Deadline: 9:00 AM, May 9, 2017**

## Exercise 1

Implement a class called `CASLock`, which implements the `java.util.concurrent.locks.Lock` interface. You are required to implement the lock and unlock methods (for the other interface required methods you can just keep them blank or returning a not significant value depending on the case). For the locking functionality make use of an `AtomicInteger` object from the types provided by the `java.util.concurrent` package in a similar fashion as the `TASLock` described during the course. The `AtomicInteger` will be considered "locked" when its value is 1 or "unlocked" when its value is 0. Refine your class as `CCASLock` in a similar way as the TTAS lock described during the course.

Compare your classes performance to the Peterson lock (the unfair version) in the same context from the last assignment - protect a shared counter incremented by the threads by using the lock, and maintain also one local counter per thread to track the accesses in the critical section. Use a counter of 300000. Perform experiments on 4 and 8 threads (without the single processor restriction) and include in a file named E1.txt statistics as in the previous assignment (case, counter value, number of increments done by each thread - the value of their local counter, number of threads, execution time). Include these statistics, along with your opinions about the results in a file named E1.txt. (Note: If you did not implement the Peterson lock for the previous assignment just skip the comparison considerations and report what you can observe regarding the performance of the two locks in this exercise).

# Exercise 2

Consider an unbounded lock based queue with the following `deq()` method:

```
public T deq() throws Exception {

        T result;

        deqLock.lock();

        try {

                if (head.next == null) {

                        System.out.println("queue empty");

                        throw new Exception();

                }

                result = head.next.value;

                head = head.next;

        } finally {

                deqLock().unlock();

        }

        return result;

}
```

Is it necessary to protect the check for a non-empty queue in the `deq()` method with a lock or the check can be done outside the locked part? Give an argument for your answer in a file Ex2.txt.

# Exercise 3

Consider the queue described in the next listing.

```
public class HWQueue <T> {

        AtomicReference <T>[] items;

        AtomicInteger tail;


 public void enq(T x) {

        int i = tail.getAndIncrement();

        items[i].set(x);

 }


 public T deq() {

        while(true) {

          int range = tail.get();

          for (int i = 0; i < range; i++) {

            T value = items[i].getAndSet(null);

             if (value != null)

               return value;

           }

         }

 }}
```

Does this algorithm always preserve FIFO order in the sense that for example if the enqueue for an element by a thread A starts before another enqueue element for a thread B, a following dequeue started by a thread C will always return the element enqueued by thread A? Detail your answer.

Please include your answers in a file Ex3.txt