Concurrency:
Multi-core Programming and Data Processing

Assignment 04

Professor: Pascal Felber
Assistant: Isabelly Rocha

April 23, 2020

Submission: Java code and a pdf file containing your answers.

1 CASLock

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 and "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 report 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 explanations for the results.

(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).

2 Unbounded lock

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

```
method T deq() throws Exception:
    T result;
    deqLock.lock();
    try:
        if (head == null) then
            System.out.println("queue empty");
            throw new Exception();
        end
        result = head.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.

3 Queue

Consider the queue described bellow.

```
class HWQueue<T>:
    AtomicReference <T>[] items;
    AtomicInteger tail;
    method void enq(T x):
        int i = tail.getAndIncrement();
        items[i].set(x);
    method T deq():
        while true do
            int i = tail.get();
            for int i = 0; i < range; i++ do
                T value = items[i].getAndSet(null);
                if value != null then
                    return value;
                end
            end
        end
```

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