CONCSYS13 - ASSIGNMENT 5

Send your solution to yaroslav.hayduk@unine.ch and emanuel.onica@unine.ch starting with [CONCSYS13] as subject. Please make a ZIP or TAR.GZ with all your files (sources, text file with explanations, ...).

Deadline: May 23rd 2013

Exercise 1

Is it necessary to protect the check for a non-empty queue in the *deq()* method of the unbounded lock based queue presented in the lab page or the check can be done outside the locked part ? Give an argument for your answer in a file Ex1.txt (you can find the listing for the queue implementation also at the end of this assignment)

Exercise 2

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;
                        }
                }
        }
}
}
```

1. Is the queue algorithm described lock-free considering the *enq()* and *deq()* methods ? Give an argument for your answer (hint: before answering recheck the lock-free description for data structures in the course).
2. 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 Ex2.txt

Exercise 3

Implement the unbounded lock-based queue and the unbounded lock-free queue described during in the lab page (listings given at the end of assignment also). Use a similar test case with the one in the previous assignment to compare the performance of your queues:

Test the queues in time measurement benchmarks on the Sun machine for sets of 2,4 and 8 threads that should work as described in the following. The share of elements to be processed by each thread will be the corresponding ratio from 100000 - for 2 threads, 50000 elements per thread; for 4 threads 25000 elements per thread; for 8 threads 12500 elements per thread (please note that considering that here you don't have to search for the position, but just to enqueue at one end and dequeue from the other, the value of the processed elements is not important). Half of the threads will try to enqueue their share and half to deque their share. Report in a file named Ex3.txt your benchmark results and what you can conclude regarding the performance difference between the two types of queue. (hint: use the barrier described in a previous lab to synchronize the thread start in order to get more significant results)

The unbounded lock-based queue (pseudocode):

```
public void enq (T value) {
        enqLock.lock();
        try {
                Node newNode = new Node(value);
                tail.next = newNode;
                tail = newNode;
        } finally {
                enqLock().unlock();
        }
}
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;
}
```

The unbounded lock-free queue (pseudocode):

```
public void enq (T value) {
        Node newNode = new Node(value);
        while (true) {
                Node last = tail.get();
                Node next = last.next.get();
                if (last == tail.get()) {
                        if (next == null) {
                                if (last.next.compareAndSet(next,node)) {
                                        tail.compareAndSet(last,node);
                                        return;
                                }
                        } else {
                                tail.compareAndSet(last,next);
                        }
                }
        }
}
public T deq() throws Exception {
        while (true) {
                Node first = head.get();
                Node last = tail.get();
                Node next = first.next.get();
                if (first == head.get()) {
                        if (first == last) {
                                if (next == null) {
                                        System.out.println("queue empty");
                                        throw new Exception();
                                }
                                tail.compareAndSet(last, next);
                        } else {
                                T value = next.value;
                                if (head.compareAndSet(first,next))
                                        return value;
                        }
                }
        }
}
```