Lecture: Concurrency Theory and Practise
Exercise 4
`http://proglang.informatik.uni-freiburg.de/teaching/concurrency/2010ws/`

1 Santa Claus and his team

Santa Claus sleeps in his shop up a the North Pole, and can only be wakened by either all nine reindeer, Dasher and Dancer, Prancer and Vixen, Comet and Cupid, and Rudolf and Blitzen, being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys.

One elf' s problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), therefore, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elve's wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't wan t to leave the tropics, and therefore the y stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise .... This could also explain the quickness in their delivering of presents , since the reindeer can't wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Define an algorithm for Santa and his team using appropriate synchronization and communication primitives (like barriers). Each participant is modeled with his own thread, i.e. there is a Santa thread, several elve threads and the reindeer threads.

1. What is the name of the ninth reindeer?
2. Design and implement the algorithm while waiting for your Christmas presents! You better make sure that Santa is not dead-locked....

Each submitted solution will earn some cookie!

2 Theory

2.1 Queues

Consider this unbounded queue implementation:

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail ;
4    ...
5    public void enq(T x) {
6      int i = tail.getAndIncrement();
7      items[i].set (x);
8    }
9    public T deq() {
10     while (true) {
11       int range = tail.get();
12       for (int i = 0; i < range; i++) {
13         T value = items[i].getAndSet(null);
14         if (value != null) {
15           return value;
16         }
17       }
18     }
19   }
20 }
```

This queue is blocking, meaning that the deq() method does not return until it has found an item to dequeue. The queue has two fields: items is a very large array (you may assume that we never address an element beyond the array bounds), and tail is the index of the next unused element in the array.

1. Are the enq() and deq() methods wait-free? If not, are they lock-free? Explain.

2. Identify the linearization points for enq() and deq(). (Careful! They may be execution-dependent.)

2.2 Stacks

Consider the problem of implementing a bounded stack using an array indexed by a top counter, initially zero. In the absence of concurrency, these methods are almost trivial. To push an item, increment top to reserve an array entry, and then store the item at that index. To pop an item, decrement top, and return the item at the previous top index.

Clearly, this strategy does not work for concurrent implementations, because one cannot make atomic changes to multiple memory locations. A single synchronization operation can either increment or decrement the top counter, but not both, and there is no way atomically to increment the counter and store a value.

Nevertheless, Bob D. Hacker decides to solve this problem. He decides to adapt the dual-data structure approach to implement a dual stack. For dual data structures, methods take effect in two stages, reservation and fulfillment. This allows waiting threads to spin on locally cached flags, and also ensures fairness in a natural way.

```
1  public class DualStack<T> {
2    private class Slot {
3      boolean full = false;
4      volatile T value = null;
5    }
6    Slot [] stack;
7    int capacity;
8    private AtomicInteger top = new AtomicInteger(0); // array index
9    public DualStack(int myCapacity) {
10     capacity = myCapacity;
11     stack = (Slot[]) new Object[capacity];
12     for (int i = 0; i < capacity; i++) {
13       stack[i] = new Slot();
14     }
15   }
16   public void push(T value) throws FullException {
17     while (true) {
18       int i = top.getAndIncrement();
19       if ( i > capacity - 1) { // is stack full?
20         throw new FullException();
```

```
21        } else if ( i > 0) { // i in range, slot reserved
22          stack[i].value = value;
23          stack[i].full = true; //push fulfilled
24          return;
25        }
26      }
27    }
28    public T pop() throws EmptyException {
29      while (true) {
30        int i = top.getAndDecrement();
31        if ( i < 0) { // is stack empty?
32          throw new EmptyException();
33        } else if ( i < capacity - 1) {
34          while (!stack [i].full){};
35          T value = stack[i].value;
36          stack[i].full = false;
37          return value; //pop fulfilled
38        }
39      }
40    }
41  }
```

Bob's DualStack<T> class splits push() and pop() methods into reservation and fulfillment steps. The stack's top is indexed by the top field, an AtomicInteger manipulated only by getAndIncrement() and getAndDecrement() calls. Bob's push() method's reservation step reserves a slot by applying getAndIncrement() to top. Suppose the call returns index i. If i is in the range 0 . . . capacity - 1, the reservation is complete. In the fulfillment phase, push(x) stores x at index i in the array, and raises the full i flag to indicate that the value is ready to be read. The value field must be volatile to guarantee that once the flag is raised, the value has already been written to index i of the array.

If the index returned from push()'s getAndIncrement() is less than 0, the push() method repeatedly retries getAndIncrement() until it returns an index greater than or equal to 0. (The index could be less than 0 due to getAndDecrement() calls of failed pop() calls to an empty stack. Each such failed getAndDecrement() decrements the top by one more past the 0 array bound. If the index returned is greater than capacity-1, push() throws an exception because the stack is full.

The situation is symmetric for pop(). It checks that the index is within the bounds and removes an item by applying getAndDecrement() to top, returning index i. If i is in the range 0 . . . capacity-1, the reservation is complete. For the fulfillment phase, pop() spins on the full flag of array slot i, until it detects that the flag is true, indicating that the push() call is successful.

What is wrong with Bob's algorithm? Is this problem inherent or can you think of a way to fix it?

2.3 Mathematical tidbits

Prove that the Bitonic[w] network has depth $(\log w)(1+\log w)/2$ and uses $(w \log w)(1+\log w)/4$ balancers.

2.4 Switching Networks

A switching network is a directed graph, where edges are called wires and node are called switches. Each thread shepherds a token through the network. Switches and tokens are allowed to have internal states. A token arrives at a switch via an input wire. In one atomic step, the switch absorbs the token, changes its state and possibly the token's state, and emits the token on an output wire. Here, for simplicity, switches have two input and output wires. Note that switching networks are more powerful than balancing networks, since switches can have arbitrary state (instead of a single bit) and tokens also have state.

An adding network is a switching network that allows threads to add (or subtract) arbitrary values.

We say that a token is in front of a switch if it is on one of the switch's input wires. Start with the network in a quiescent state q_0, where the next token to run will take value 0. Imagine we have one token t of weight a and n-1 tokens $t_1, \ldots, t_{n-1}$ all of weight b, where $b > a$, each on a distinct input wire. Denote by S the set of switches that t traverses if it traverses the network by starting in q_0.

Prove that if we run the $t_1, \ldots, t_{n-1}$ one at a time though the network, we can halt each t_i in front of a switch of S. At the end of this construction, $n-1$ tokens are in front of switches of S. Since switches have two input wires, it follows that t's path through the network encompasses

at least $n-1$ switches, so any adding network must have depth at least $n-1$, where n is the maximum number of concurrent tokens. This bound is discouraging because it implies that the size of the network depends on the number of threads (also true for CombiningTrees, but not counting networks), and that the network has inherently high latency.

2.5 Skip Lists

Consider this buggy contains method for the LockFreeSkipList class.

```
1  boolean contains(T x) {
2    int bottomLevel = 0;
3    int key = x.hashCode();
4    Node<T> pred = head;
5    Node<T> curr = null;
6    for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
7      curr = pred.next[level].getReference();
8      while (curr.key < key ) {
9        pred = curr;
10       curr = pred.next[level].getReference();
11     }
12   }
13   return curr.key == key;
14 }
```

Give a scenario where this method returns a wrong answer.

3 Practise

3.1 Keep counting!

Implement a trinary CombiningTree, that is, one that allows up to three threads coming from three subtrees to combine at a given node.

What are the advantages and disadvantages of such a tree when compared to a binary combining tree?

3.2 Balancing things out

Provide an efficient lock-free implementation of a Balancer with traverse() and antiTraverse() methods.

If this was too easy, provide also a wait-free implementation.

Submission

- Deadline: 11.01.2011
- Please submit solutions in teams of 2 or 3 people. (Submission of single-person teams will not be corrected!)