**Chalmers** | GÖTEBORGS UNIVERSITET
Computer Science and Engineering

# Principles of Concurrent Programming TDA384/DIT391

Tuesday, 19 December 2017

(including example solutions)

**Teacher/examiner:** K. V. S. Prasad (prasad@chalmers.se)

## Exercise 1: Concurrent data structures *(14 points)*

In this exercise, you will evaluate different implementations of a counting operation on a list data structure in Java, analyzing whether they are *thread safe*, that is executable by multiple concurrent threads without running into race conditions.

Recall the various implementations of linked sets presented during the course (and also described in Chapter 9 of Herlihy & Shavit). They are all variants implementing the same *interface*, consisting of operations to remove elements, add elements, and test whether an element is in the set. In this exercise, we extend the interface with a new operation `int size()`, which simply returns the number of nodes stored in the set.

A simple implementation of method `size()` that works in a sequential setting is:

```
public int size() {
  int size = -1;
  Node<T> curr;
  curr = head;              // set curr to the head node
  do {
    curr = curr.next();     // move curr to next node in chain
    size += 1;              // increment size by 1
  } while (curr != tail);  // until curr reaches the tail node
  return size;
}
```

**Question 1.1 (2 points):** Why is local variable `size` initialized to `-1` instead of `0`?

The empty set consists of the `head` node, whose `next` node is the `tail` node. Hence, `size` is incremented at least once, returning `0` for an empty set as it should be.

**Question 1.2 (2 points):** Explain why the above implementation of `size()` is not thread safe. To this end, describe a concrete scenario where race conditions may occur.

Without any kind of concurrency control, `size()` may interfere with other threads executing destructive operations on the list. For example, a thread $t$ may be removing an element from the set;

the result of another thread $u$ calling to `size()` depends on whether $t$ is operating in the portion of the linked list already scanned by $u$: if $t$ removes an element behind $u$'s `curr`, $u$ will include it in the count even if $t$ terminates before $u$.

**Question 1.3 (2 points):** Recall the implementation `CoarseSet` of the thread-safe set data structures seen during the course (and also described in Chapter 9 of Herlihy & Shavit). `CoarseSet` uses a variable `lock` of type `Lock` to guard access to the whole data structure. Modify the implementation of `size()` shown above so that it uses `lock` to avoid race conditions.

Since `lock` guards access to the whole list, it is sufficient to acquire `lock` at the beginning of `size()` and to release it at the end:

```
public int size() {
  lock.lock();
  try {
    // body of size() as above
  } finally {
    lock.unlock();
  }
}
```

**Question 1.4 (4 points):** In order to make `size()` run in constant time, we now consider a more efficient implementation that adds an attribute `size` of type `int` to the set, which keeps track of the current number of elements in the list. Thus, method `size()` simply returns the value of attribute `size` when it is called.

a) Write an implementation of `size()` in `CoarseSet` that is thread safe using locks.

```
public int size() {
  lock.lock();
  try {
    return size;
  } finally {
    lock.unlock();
  }
}
```

b) Which operations of `CoarseSet` must update the value of attribute `size`?

   The operations that change the set's size: `add` and `remove`.

c) Consider the implementation of method `remove` in `CoarseSet`. Illustrate what race conditions may occur if `remove` updates the value of attribute `size` *after* releasing `lock`.

   The same race condition described in the answer to question 1.2 above can occur here if the thread executing `remove` interleaves with the one executing `size()`.

**Question 1.5 (4 points):** Consider yet another variant of set implementation where we want to update attribute `size` without using any locks.

a) Choose a suitable *type* for attribute `size` so that it can be updated thread-safely without using locks.

```
AtomicInteger size;
```

b) Based on your choice of type, write a piece of code that increments `size` by one in a thread-safe manner without locking.

```
int curSize;
do {
  curSize = size.get();
} while (!size.compareAndSet(curSize, curSize + 1));
```