P. Thiemann, A. Bieniusa, P. Heidegger Winter term 2010/11

Lecture: Concurrency Theory and Practise
Exercise 3
http://proglang.informatik.uni-freiburg.de/teaching/concurrency/2010ws/

1 Theory

1.1 Linearization points

Consider this queue implementation whose enq() method does not have a linearization point.

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail ;
4    static final int CAPACITY = 1024;
5
6    public HWQueue() {
7      items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
         CAPACITY);
8      for ( int i = 0; i < items.length ; i++) {
9        items [i] = new AtomicReference<T>(null);
10     }
11     tail = new AtomicInteger(0);
12   }
13   public void enq(T x) {
14     int i = tail.getAndIncrement();
15     items[i].set (x);
16   }
17   public T deq() {
18     while (true) {
19       int range = tail.get();
20       for (int i = 0; i < range; i++) {
21         T value = items[i].getAndSet(null);
22         if (value != null) {
23           return value;
24         }
25       }
26     }
27   }
28 }
```

The queue stores its items in an items array, which for simplicity we will assume is unbounded. The tail field is an AtomicInteger, initially zero. The enq() method reserves a slot by incrementing tail , and then stores the item at that location. Note that these two steps are not atomic: there is an interval after tail has been incremented but before the item has been stored in the array. The deq() method reads the value of tail , and then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps null with the current contents, returning the first non-null item it finds. If all slots are null , the procedure is restarted.

Give an example execution showing that the linearization point for enq() cannot occur in line 14. (Hint: give an execution where two enq() calls are not linearized in the order they execute Line 14.)

Give another example execution showing that the linearization point for enq() cannot occur at Line 15.

Since these are the only two memory accesses in enq(), we must conclude that enq() has no single linearization point. Does this mean enq() is not linearizable?

1.2 Reentrant Reader-Writer Locks

The ReentrantReadWriteLock class provided by the java. util .concurrent.locks package does not allow a thread holding the lock in read mode to then access that lock in write mode (the thread will block). Justify this design decision by sketching what it would take to permit such lock upgrades.

1.3 Fine-grained linked lists

Explain why the fine-grained locking algorithm for linked lists is not subject to deadlock.

1.4 Fine-grained testing for elements

Provide the code for the contains() method missing from the fine-grained algorithm. Explain why your implementation is correct.

1.5 More on linked lists

Would the lazy algorithm still work if we marked a node as removed simply by setting its next field to null? Why or why not? What about the lock-free algorithm?

2 Practice

2.1 Adding functionality to collections

Reusing Java libraries is often preferable to creating new ones. But often the best we can find is a class that supports *almost* all the operations we want. In this case we need to add a new operation to it without undermining its thread-safety.

Implement a thread-safe List with an atomic putIfAbsent() method.

Hint: Synchronising on the List implementations that come with Java's collection classes nearly does the job, as they provide contains and add methods which you can reuse to construct the missing method. You can for example use the ArrayList class for the actual list implementation.

2.2 Pool party!

Creating a new thread for each task in a program can lead to major performance issues as thread spanning is expensive. Thread pools limit the number of threads and allow thus a good capacity utilization.

Fill an array of size n (random value between 0 and $100,000$) with random numbers from 0 to n. For each number, count its frequency in the list. To increase the application's performance, the counting tasks are to be done in different threads.

Implement a thread factory for the counting threads to be used with Java's ThreadPoolExecutor . Further, extend the ThreadPoolExecutor with statistics about the total execution time, number of tasks done, average execution time of a task, ...

2.3 Simple Numerics

Implement a class for vectors of size n (great n, e.g. $n > 10000$), $\vec{x} = (x_1, \dots, x_n)$ with $x_i \in \mathbb{R}$, as they are used in numerical computations. It should provide parallel implementations of the following methods:

- summation of all vector entries: $sum(\vec{x}) = x_1 + \cdots + x_n$
- addition of two vectors: $add(\vec{x}, \vec{y}) = (x_1 + y_1, \dots, x_n + y_n)$
- scalar multiplication: $scal(a, \vec{x}) = (ax_1, \dots, ax_n)$
- length of a vector: $length(\vec{x}) = \sqrt{x_1^2 + \cdots + x_n^2}$
- dot product of two vectors: $prod(\vec{x}, \vec{y}) = x_1y_1 + \cdots + x_ny_n$

For simplicity, you may assume that $n = 2^k$ for some $k \in \mathbb{N}$.

Submission

- Deadline: 14.12.2010
- Please submit solutions in teams of 2 or 3 people. (Submission of single-person teams will not be corrected!)