Universitetet i Oslo
Institutt for Informatikk
PMA
Olaf Owe, Martin Steffen, Toktam Ramezani

INF 4140: Models of Concurrency

Høst 2016

Mandatory assignment 1

1. Sep. 2018

Issued: 1. Sep. 2018

Due: 30. Sept. 2016 (23:59)

1 General remarks

Language

You must use English.

How to deliver

- Your solution should be delivered online (`https://devilry.ifi.uio.no`).
- Program examples should be commented in order to make them understandable for the group teacher or lecturer.

Who delivers

We encourage to work together in groups of 3 (but not more). For "technical" reasons (devilry): in a group each member should upload the same solution (which should be identical, just the same PDF uploaded several times).[1] The solution must be marked with names and email addresses of all contributing students.

Check in time that devilry works and that your status within devilry (and student web etc) is OK. Don't try your INF4140-devilry access as late as the day of the deadline. In case of doubts, for clarifications, or if having trouble with devilry etc, ask in time.

Evaluation

This assignments are graded *pass* or *fail*. You must pass the obligs in order to take the final exam.

[1] That facilitates managing acceptance.

"Thread-safe" queue as linked list

Motivation

Programming languages typically come with *libraries,* i.e., repositories of data structures and their access routines/methods (think of the Java standard libraries). Typical simple structures are lists, queues, collections, various forms of trees, but also window panels etc. Data structures may be *thread-safe* or not. Threads safety means that the data structure is implemented in such a way (using appropriate synchronization internally), that it can be used by a concurrent program providing the "expected" functionality (for instance FIFO for a queue) and without weird, sporadic errors. Sometimes, libraries provide a thread-safe and a non-thread-safe version of the same data structure. When programming concurrently, it's necessary to check the library's API documentation, whether or not the intended data structure is thread-safe. If not, of course, the programmer will typically have to take extra (synchronization) measures to make correct use of the data structure.[2]

The task described below is not about *using* an appropriate thread-safe/unsafe data structure from some library, but about implementing one oneself.

Task

A queue is often represented using a linked list. Assume that two variables, head and tail, point to the first and last elements of the queue. A null link is represented by the constant null.

Each element in the queue contains a data field (val) and a link to the next element of the queue (next). Each element in the queue is thereby represented by two variables in the program. For convenience, we use dot-notation and write el.val and el.next for the variables representing the values val and next for the queue element el. In the initial state, head = tail = null.[3]

The routine find(d) finds the first element of the queue containing the data value d.

```
1   find(d) {
2     i := head;    # variable i is local to
3                   # the current method instance
4     while (i≠ null and i.val ≠ d) { i := i.next; }
5   }
```

The routine insert (new) inserts a new element at the end of the queue. Both the head and tail pointer must be updated if the queue is empty. (Assume that new.next = null.)

```
1  insert(new) {
2    if (tail = null) {    # empty list
3      head := new;
4    } else {
5      tail.next := new;
6    }
7    tail := new
8  }
```

[2]You may read e.g. `http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html` for some typical API information; the example is for illustration only, it's not connected to the programming task of the oblig.

[3]We use the "slides" notation here: "=" is equality, ":=" is assignment. In C-like languages, == and = is common.

The routine delfront deletes the first element of the queue (pointed to by head). The variable tail must be updated if the queue contains only one element.

```
1  bool del_lock := true
2    delfront {
3      if (head ≠ null) {                  # list not empty
4        if (head = tail) tail := null; # only 1 elem. in list
5        head := head.next;
6   }
```

Do the following:

1. Identify the $\mathcal{V}$ and $\mathcal{W}$ sets (see lecture slides) of the shared variables in the three routines. You may use dot-notation in the variable representation for val and next. For example: tail.next is in the $\mathcal{W}$ set of the routine insert.[4]

2. Now assume that several processes access the linked list. Consider all six pairs combining two of three routines given above. Which combinations of routines can be executed in parallel without interference? Which combinations of routines must be executed one at a time? Remember to consider the parallel execution of each routine against itself.

   Hint: Remember that two *disjoint* routines A and B do not interfere with each other. If A and B are not disjoint, we may use the "At-Most-Once Property" to decide if A and B interfere. When interpreted on routines, "At-Most-Once Property" may be formulated as follows: Consider all possible results of executing A and B sequentially (i.e. A; B and B; A). The routines A and B interfere with each other if parallel execution of A and B may lead to any *new results* not possible from a sequential execution.

3. Add **await** statements to program the synchronization code in the routines such that non-interference of concurrent executions is enforced. Try to make your atomic actions as small as possible, and do not delay a routine unnecessarily.

   Hint: It might be helpful to use more than one lock in order to rule out the interfering combinations of executions.

[4] The definition on the slides don't cover cases like tail.next, we mentioned there only proper variables. tail is a variable, whereas technically tail.next is not. So, just use the concepts of the lecture to appropriately cover "variables" in dot notation. After all: like variables, tail.next refers to a memory address which may or may not be shared.