Universitetet i Oslo
Institutt for Informatikk

Research group for Reliable Systems (PSY)
Einar Broch Johnsen, Silvia Lizeth Tapia Tarifa, Tobias John, Juliane Päßler

INF 5170: Models of Concurrency

Fall 2024

Mandatory assignment 1

28. Aug. 2024

Due: Monday 10. Sept. 2024 (23:59 Norwegian time)

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.

Who delivers

We encourage you 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). 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 IN5170-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.

"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 is 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.

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

Problem description

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.[1]

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     # return i
6   }
```

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  }
```

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

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

Your task

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 .

2. Now assume that several processes access the linked list. Consider all six pairs combining two of three routines given above. Remember to consider the concurrent execution of each routine against itself.

   (a) Which combinations of routines can be executed concurrently without interference? Explain your reasoning, see also the hint and the AMO property below.

   (b) Which combinations of routines must be executed one at a time? Explain your reasoning by giving an example execution where the AMO property for routines (see below) does not hold.

   Hint: Remember that two routines A and B do not interfere with each other if

   $$\mathcal{V}_A \cap \mathcal{W}_B = \mathcal{V}_B \cap \mathcal{W}_A = \emptyset. \tag{1}$$

   If two routines do not satisfy 1, then we have to check the At-Most-Once Property interpreted for routines to check if the routines can be executed concurrently.

   At-Most-Once (AMO) Property for routines: Two routines A and B satisfy the AMO Property if concurrent execution of A and B does *not* lead to new results compared to their sequential execution (i.e., $A;B$ and $B;A$).

3. The last task is to implement a simplified map-reduce system that computes the sum of all squares up to a certain integer, using a thread-safe linked queue and thread pools. The system has the following components:

   - An input queue, that has to be filled with the first n numbers.
   - A queue for all even numbers.
   - A queue for all odd numbers.
   - Two *mappers*: each mapper takes one number from the input queue and adds its square to the even queue (if the number is even) or to the odd queue (if the number is odd).
   - Two *reducers*: each reducer is assigned one queue (either odd or even) and computes the sum of all elements.
   - One thread pool to fill the input queue

- One thread pool to take values from the input queue and distribute them to the two mappers
- One thread pool to take values from the even and odd queue and transmit them to the correct reducer (the even numbers to the "even reducer", the odd numbers to the "odd reducer").

Under `https://www.uio.no/studier/emner/matnat/ifi/IN5170/h24/obligs/` you will find a skeleton of a Java application that realizes this architecture. Please fill in the marked places in the source code to realize the above behavior and add modifiers to fields, classes and methods as necessary to ensure thread safety. You should pay attention to the following.

- Submit executable code.
- The method find should return the pointer to the node in the queue with content t, not return the node.
- Synchronize where you have to do so.
- Don't synchronize where you don't have to! For example, the three tasks (inserting in the input queue, mapping and reducing) should be done concurrently.
- Both mappers should insert in the even as well as in the odd queue.