COP 5618/CIS 4930 HW2

Due: Nov 7 at 11:50pm

Suppose that there are three kinds of operations on singly-linked list: search, insert and remove. Searching threads merely examine the list; hence they can execute concurrently with each other. Inserting threads add new items to the front of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, the remove operation removes items from anywhere in the list. At most one thread can remove items at a time, and a remove must also be mutually exclusive with searches and inserts.

Start with the skeleton class in ConcurrentSearcherList.java. Your task is to modify the class to allow access by multiple threads according to the description above. To help you design your solution, answer questions 1-4 as a pencil and paper exercise:

1. Which variables will you add and what is their interpretation and initial value?
   a. nS //number of active searchers, initially 0
   b. nI //number of active inserters, initially 0
   c. nR //number of active removers, initially 0
2. Give an invariant that captures the synchronization requirements above? Give your invariant as a logical formula and express it in English. (Hint: Make sure that your invariant allows for no threads accessing the list. You may recall from discussions of student solutions in class when we were looking at the readers/writers problem that forgetting this corner case is a common error.)

   $0 \le nS$ and $0 \le nI \le 1$ and $0 \le nR \le 1$ and $(nR==0$ or $(nS==0$ and $nI==0))$

   The number of searchers is at least 0, the number of inserters is at least zero, and there can be at most one inserter at a time, the number of removers is at least zero, and there can be at most one remover at a time. Also, either there are no removers (allowing searchers and inserters to be active) or there are no searchers and no inserters.

3. Provide implementations for start_search, end_search, start_insert, end_insert, start_remove, and end_remove *in terms of atomic_await statements*.

```
void start_insert()<
      await (nremovers == 0 && ninserters == 0))
      ninserters++;
>

void end_insert()<
      ninserters--;
>

void start_search()<
      await (nremovers == 0)
```

```
        nsearchers++;
    >

    void end_search()<
        nsearchers--;
    >

    void start_remove() <
        await  (nsearchers == 0 && ninserters == 0 && nremovers == 0)
        nremovers++;
    >

    void end_remove() <
        nremovers--;
    >
```

4. The start_search, and start_insert methods should allow an inserter to execute concurrently with searchers. If you convert your atomic await solution to a legal Java implementation are there conflicting operations within the provided bodies of the search and insert methods? Do these conflicts create data races? Are new Nodes created by the insert method safely published?

   The invariant allows conflicting operation, specifically the modification of first in insert, and the read of first in search. These conflicts are not ordered by happens before by the standard atomic await implementation. Because new Node instances are published (i.e. made visible to other threads) by writing their address to first, new Node instances are not safely published. Making first volatile solves the problem. (Changing first to an AtomicReference would also work but would have required changes everywhere in the code that first is accessed.)

5. Modify the given class in accordance with your answers above. You may add additional variables, add keywords, and modify the body of the constructor (but not its signature). You should not need to modify class Node. There is a solution that does not require modification of the public methods insert, remove, and search. Your solution should not have data races.
6. (COP5618 students only) Think about whether this class is likely to be useful in practice. Briefly discuss its pros and cons.

   Pros: Allows maximal reasonable concurrency.

   Cons: the granularity of actions is probably too small to be useful. What benefit is there to knowing that a value is an element of the list if another thread can remove it at any time. Our modifications allow safe concurrent access to the list in a way that ensures there are no data races and preserve the integrity of the list itself. Whether the results are useful without additional synchronization by the application is questionable.

You should, of course, create tests for your solution. However, in this assignment we will not collect or evaluate your test cases.

Turn in:

1. PDF file containing your answers to Questions 1-4, and 6 if applicable.
2. One uncompressed Java file called ConcurrentSearcherList.java containing your solution for Question 5.