Handout 3
Concurrent Queues with Stacks

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

17 May, 2022

This project is due on Friday, June 2, at 23h59m. The exact details on how to turn in your solution will be made available at a later date. You will be expected to submit a zip file with your (annotated) Java files that should pass Verifast's analysis (with the arithmetic overflow check turned **off**). All methods must have pre and post conditions. Code that fails to meet these requirements will be considered as a failing project.

The project consists of verifying a concurrency-safe queue, implemented with two stacks (see description below), using the techniques discussed in class. The project is split into two tasks: Task 1 (Section 1) consists of the implementation and verification of a sequential stack; Task 2 (Section 2) consists of the implementation and verification of the concurrency-safe queue. You will have to install the spec `java.util.concurrent.locks.javaspec` (available online in CLIP) for the package `java.util.concurrent.locks` by copying it to the `bin/rt/` folder of your local verifast installation and append the `rt.jarspec` file with the name of the spec file.

Important Note: inconsistent predicates used as invariants or in preconditions of methods will let you **incorrectly** verify code. The symptom will be that you will be able to prove false, and that you will not be able to call those methods (since it is impossible to satisfy the precondition).

1 Sequential Stack (Task 1)

For this task we will not (yet) be concerned with concurrency. The goal here is to implement an unbounded stack. The stack will be backed by a chain of singly-linked `Node` objects. The stack must support the following API (you will need to use the `Node` class presented in lecture):

```
public class Stack {
  ...

  public Stack() { ... }

  public boolean isEmpty() { ... }

  public void push() { ... }

  public int pop() { ... }
```

```
    public int peek() { ... }

    public void flip() { ... }

}
```

Constructor. The constructor simply initializes an empty stack.

isEmpty and peek. The `isEmpty` method returns **`true`** if and only if the stack is empty. The `peek` method, which can only be called on a non-empty stack, returns the element at the top of the stack.

push and pop. The `push` method inserts the given element on the top of the stack. The `pop` method, which can only be called if the stack is not empty, removes the top element of the stack and returns it.

flip. The flip method inverts the order of the elements of the stack.

1.1 Verification

The specification of the stack should be represented as one (or more) predicates that characterize the memory footprint of a stack and, crucially, that **maintain the logical sequence of integers that the stack codifies** (as a logical list of integers). The specifications of the individual methods of the stack should precisely characterize the method's return values (when they exist), as well as, the elements in the stack in pre- and post-conditions via the logical list.

Hint: For the specification and verification of `flip()` study the lecture on verification of pointer-based data structures.

2 Concurrent Queue (Task 2)

The concurrency-safe FIFO queue should make use of your stack implementation of **Task 1**, adhering to the following structure:

```
public class CQueue {

    Stack left;
    Stack right;
    //Other relevant fields go here...

    public CQueue() { ... }

    private void enqueue(int elem) { ... }

    public void flush() { ... }

    public int dequeue() { ... }
```

```
    public boolean isEmpty() { ... }
}
```

The FIFO Queue's operations must be implemented (using monitors and conditions) such that they can be **safely** used by concurrent threads, as discussed in lecture.

In terms of **sequential** functionality, the queue relies on two stacks to provide amortized $\mathcal{O}(1)$ enqueue and dequeue operations. The idea is that enqueues are performed by pushing on the `left` stack, whereas dequeues are performed by popping from the `right` stack. Since the stacks are of unbounded capacity, the enqueue operation can always perform the respective push on `left`. However, the dequeue operation must first test whether the `right` stack is empty. If so, it calls the `flush()` method, which moves elements from the `left` to the `right` stack (in reverse order) after which it can now perform the dequeue by popping from the `right` . Naturally, the dequeue operation can only be called on a non-empty queue.

The constructor initializes an empty queue, initializing the two stacks. The `enqueue` and `dequeue` operations add and remove elements from the queue in FIFO order, according to the strategy discussed above. The `flush` private method, which can only be called when the `left` stack is not empty and the `right` stack is empty, switches the elements of the `left` stack to the `right` stack (in reverse order).

2.1 Client Code

You will also be required to implement a client that launches 100 threads to **(a)** enqueue elements; **(b)** dequeue elements from the LIFO; printing a log on the standard output. All threads should run concurrently.

2.2 Verification

You will need to use the verification technique for monitors and conditions discussed in class to verify concurrent usages of the FIFO Queue. It will be convenient to define **two predicate constructors**: one for the shared queue state and a specialized variant that relates to the conditions necessary to verify the operations of the queue. These invariants should reuse the infrastructure from **Task 1**. You will also need to define the **concurrent invariant**, which specifies the memory layout of the concurrent sequence and the logical representation of any monitors and conditions. This invariant must be preserved by all the operations of the queue.

Note that although the concurrent queue invariant and the shared state invariants do not mention the logical queue contents, the specification of the private method `flush` (which is only called when the lock is held in the `dequeue` method) will require mentioning the inversion of the order of the elements in one of the stacks. `Hint:` The code for `flush` should not rely on loops. Check the Stack API carefully.

In the verification of the `dequeue()` method, it will be convenient to write code that is somewhat "unnatural", for the purposes of simplifying the verification. Specifically, you will want to test whether the `right` stack is empty and if so, `flush` **and** pop from `right`; otherwise, `pop` from `right`. One of the following lemmas may also prove useful when checking the **`then`** branch described above (add them to the start of the file):

```
lemma void append_nnil(list<int> l1, list<int> l2)
    requires l2 != nil;
    ensures append(l1,l2) != nil;
{
```

```
    switch (l1) {
        case nil:
        case cons(x,xs):

    }
}
lemma void reverse_nnil(list<int> xs)
    requires xs != nil;
    ensures reverse(xs) != nil;
    {
        switch(xs) {
            case nil:
       case cons(h, t):
        append_nnil(reverse(t),cons(h,nil));

      }
  }
```

3 Grading

The two tasks will be split as follows: Task 1 will be 60% of the grade; Task 2 will be 40% of the grade. You **must** adhere to the specification requirements described in the handout.