

3. Exercises: Basic building blocks of concurrency

DCC-FCUP, University of Porto

José Proença



Concurrent Programming – Part 2

These exercises are taken mainly from the book “*Learning Concurrent Programming in Scala*”. Most of these require implementing new concurrent data structures using atomic variables and the CAS instruction, although they can also be solved using the synchronized statements.

Exercise 1. Implement a custom `ExecutionContext` class¹ called `PiggybackContext`, which executes `Runnable` objects on the same thread that calls the `execute` method. Ensure that a `Runnable` object executing on the `PiggybackContext` can also call the `execute` method and that exceptions are properly reported.

Exercise 2. Implement a `TreiberStack` class, which implements a concurrent stack abstraction:

```
class TreiberStack[T] {  
  def push(x: T): Unit = ???  
  def pop(): T = ???  
}
```

Use an atomic reference variable that points to a linked list of nodes that were previously pushed to the stack. Make sure that your implementation is lock-free and not susceptible to the ABA problem.

Exercise 3. Implement a `ConcurrentSortedList` class, which implements a concurrent sorted list abstraction:

```
class ConcurrentSortedList[T](implicit val ord: Ordering[T]) {  
  def add(x: T): Unit = ???  
  def iterator: Iterator[T] = ???  
  ...  
  case class Node(head: T, tail: AtomicReference[...])  
}
```

Under the hood, the `ConcurrentSortedList` class should use a (manually created) linked list of atomic references by inserting elements in the right position. Ensure that your implementation is lock-free and avoids ABA problems. The `Iterator` object returned by the `iterator` method must correctly traverse the elements of the list in ascending order under the assumption that there are no concurrent invocations of the `add` method.

Exercise 4. If required, modify the `ConcurrentSortedList` class from the previous example so that calling the `add` method has the running time linear to the length of the list and creates a constant number of new objects when there are no retries due to concurrent `add` invocations.

Exercise 5. Implement a `LazyCell` class with the following interface:

¹<https://www.scala-lang.org/api/2.13.3/scala/concurrent/ExecutionContext.html>

```
class LazyCell[T](initialization: =>T) {  
  def apply(): T = ???  
}
```

Creating a `LazyCell` object and calling the `apply` method must have the same semantics as declaring a lazy value and reading it, respectively. You are not allowed to use lazy values in your implementation. Avoid calling `synchronized` in a normal execution (i.e., without data races).

Exercise 6. Implement a `PureLazyCell` class with the same interface and semantics as the `LazyCell` class from the previous exercise. The `PureLazyCell` class assumes that the initialization parameter does not cause side effects, so it can be evaluated more than once. The `apply` method must be **lock-free** and should call the initialization as little as possible.

Exercise 7. Implement a `SyncConcurrentMap` class that extends the `Map` API that can be found in the `scala.collection.concurrent` package.² Use the `synchronized` statement to protect the state of the concurrent map, captured by a traditional mutable map (from `scala.collection.mutable.Map`).

²<https://www.scala-lang.org/api/2.13.x/scala/collection/concurrent/Map.html>