

## 2. Exercises: Concurrency in Java and its memory model

DCC-FCUP, University of Porto

José Proença



Concurrent Programming – Part 2

These exercises are taken mainly from the book “*Learning Concurrent Programming in Scala*”, and are designed to test the knowledge of the Scala programming language. You are required to implement higher-level concurrency abstractions in terms of basic JVM concurrency primitives. The exercises are not ordered in any particular order, but some assume earlier exercises have been done.

**Exercise 1.** Implement a `parallel` method, which takes two computation blocks, `a` and `b`, and starts each of them in a new thread. The method must return a tuple with the result values of both the computations. It should have the following signature:

```
def parallel[A, B](a: =>A, b: =>B): (A, B)
```

Place the `thread` and the `log` function, defined in the theoretical lessons, in a package object and use them when defining `parallel`.

**Exercise 2.** Implement a `periodically` method, which takes a time interval duration specified in milliseconds, and a computation block `b`. The method starts a thread that executes the computation block `b` every duration milliseconds. It should have the following signature:

```
def periodically(duration: Long)(b: =>Unit): Unit
```

Extra: try to use a daemon thread.

**Exercise 3.** Implement a `SyncVar` class with the following interface:

```
class SyncVar[T] {  
  def get(): T = ???  
  def put(x: T): Unit = ???  
}
```

A `SyncVar` object is used to exchange values between two or more threads. When created, the `SyncVar` object is **empty**:

- calling `get` throws an exception, and
- calling `put` adds a value to the `SyncVar` object.

After a value is added to a `SyncVar` object, we say that it is **non-empty**:

- calling `get` returns the current value, and changes the state to empty, and
- calling `put` throws an exception.

**Exercise 4.** The `SyncVar` object from the previous exercise can be cumbersome to use, due to exceptions when the `SyncVar` object is in an invalid state. Implement a pair of methods, `isEmpty` and `nonEmpty`, on

the `SyncVar` object. Then, using these, implement a **producer thread** that transfers a range of numbers 0 until 15, and a **consumer thread** that receives these (using busy-waiting) and prints them. Run both threads in parallel.

**Exercise 5.** Using the `isEmpty` and `nonEmpty` pair of methods from the previous exercise requires busy-waiting. Add the following methods to the `SyncVar` class:

```
def getWait(): T
def putWait(x: T): Unit
```

These methods have similar semantics as before, but go into the waiting state instead of throwing an exception, and return once the `SyncVar` object is empty or non-empty, respectively.

**Exercise 6.** A `SyncVar` object can hold at most one value at a time. Implement a `SyncQueue` class, which has the same interface as the `SyncVar` class, but can hold at most `n` values. The `n` parameter is specified in the constructor of the `SyncQueue` class.

[ *Suggestion: use the `scala.collection.mutable.Queue` data structure.* ]

**Exercise 7.** The `send` method in the Deadlocks' slides was used to transfer money between the two accounts. The `sendAll` method takes a set accounts of bank accounts and a target bank account, and transfers all the money from every account in `accounts` to the target bank account. The `sendAll` method has the following signature:

```
def sendAll(accounts: Set[Account], target: Account): Unit
```

Implement the `sendAll` method and ensure that a deadlock cannot occur.

**Exercise 8.** Recall the `asynchronous` method from the Guarded blocks' slides. This method stores the tasks in a First In First Out (FIFO) queue; before a submitted task is executed, all the previously submitted tasks need to be executed. In some cases, we want to assign priorities to tasks so that a high-priority task can execute as soon as it is submitted to the task pool. Implement a `PriorityTaskPool` class that has the `asynchronous` method with the following signature:

```
def asynchronous(priority: Int)(task: =>Unit): Unit
```

A single worker thread picks tasks submitted to the pool and executes them. Whenever the worker thread picks a new task from the pool for execution, that task must have the highest priority in the pool.

[ *Suggestion: use the `scala.collection.mutable.PriorityQueue` data structure.* ]

**Exercise 9.** Extend the `PriorityTaskPool` class from the previous exercise so that it supports any number of worker threads `p`. The parameter `p` is specified in the constructor of the `PriorityTaskPool` class.

**Exercise 10.** Extend the `PriorityTaskPool` class from the previous exercise so that it supports the shutdown method:

```
def shutdown(): Unit
```

When the `shutdown` method is called, all the tasks with the priorities greater than `important` must be completed, and the rest of the tasks must be discarded. The `important` integer parameter is specified in the constructor of the `PriorityTaskPool` class.

**Exercise 11.** Implement a `ConcurrentBiMap` collection, which is a concurrent bidirectional map. The invariant is that every key is mapped to exactly one value, and vice versa. Operations must be atomic. The concurrent bidirectional map has the following interface:

```
class ConcurrentBiMap[K, V] {  
  def put(k: K, v: V): Option[(K, V)]  
  def removeKey(k: K): Option[V]  
  def removeValue(v: V): Option[K]  
  def getValue(k: K): Option[V]  
  def getKey(v: V): Option[K]  
  def size: Int  
  def iterator: Iterator[(K, V)]  
}
```

Make sure that your implementation prevents deadlocks from occurring in the map.

**Exercise 12.** Add a `replace` method to the concurrent bidirectional map from the previous exercise. The method should atomically replace a key-value pair with another key-value pair:

```
def replace(k1: K, v1: V, k2: K, v2: V): Unit
```

**Exercise 13.** Test the implementation of the concurrent bidirectional map from the earlier exercise by creating a test in which several threads concurrently insert millions of key-value pairs into the map. When all of them complete, another batch of threads must concurrently invert the entries in the map – for any key-value pair `(k1, k2)`, the thread should replace it with a key-value pair `(k2, k1)`.