Exercise 1 : Introduction to Concurrency

Freshly graduated from EPFL, you all have been hired as contractors for a successful and rapidly growing bank. The bank has recently been experiencing problems with their money management system, coded in Scala, and so they hired the best and brightest young engineers they could find: you! The system has been working perfectly fine so far, they tell you. In the past days, due to an increased number of customers, they had to switch from a single threaded sequential execution environment to a multithreaded concurrent environment, in which multiple threads may perform transactions concurrently. That's when problems started, your manager says…

Below is the code responsible to withdraw money from the account `from` and transfer it to the account `to`, within the same bank.

```
def transfer(from: Account, to: Account, amount: BigInt) {
  require(amount >= 0)

  val balanceFrom = from.balance
  if (balanceFrom >= amount) {
    from.balance = balanceFrom - amount
    val balanceTo = to.balance
    to.balance = balanceTo + amount
  }
}
```

For the bank, it is very important that the following two properties hold in any sequence of transfer transactions:

1. The balance of an account never goes below 0.
2. The total sum of money held by the bank is constant.

Question 1

Does the above `transfer` method respect the two properties in a **sequential** execution environment, that is, when there is only one thread in the program?

Question 2

What can go wrong in a setting where multiple threads can execute the `transfer` method concurrently ? For each of the two desired properties of the system, check if its holds in this concurrent environment. If not, come up with an example execution which exhibits a violation of the property.

Question 3

For each of the proposed implementations of `transfer` below, check which of the properties hold. Additionally, check if the system is vulnerable to *deadlocks*.

Variant 1

```
def transfer(from: Account, to: Account, amount: Long) {
  val balanceFrom = from.balance
  if (balanceFrom >= amount) {
    from.synchronized {
      from.balance = balanceFrom - amount
    }
    to.synchronized {
      val balanceTo = to.balance
      to.balance = balanceTo + amount
    }
  }
}
```

Variant 2

```
def transfer(from: Account, to: Account, amount: Long) {
  from.synchronized {
    val balanceFrom = from.balance
    if (balanceFrom >= amount) {
      from.balance = balanceFrom - amount
      to.synchronized {
        val balanceTo = to.balance
        to.balance = balanceTo + amount
      }
    }
  }
}
```

Variant 3

```
object lock // Global object.
def transfer(from: Account, to: Account, amount: Long) {
  lock.synchronized {
    val balanceFrom = from.balance
    if (balanceFrom >= amount) {
      from.balance = balanceFrom - amount
      val balanceTo = to.balance
      to.balance = balanceTo + amount
    }
  }
}
```

Exercise 2 : Parallel Reductions

Question 1

As a group, write a function called `minMax`, which should take a non-empty array as input and return a pair containing the smallest and the largest element of the array.

```
def minMax(a: Array[Int]): (Int, Int) = ???
```

Now write a parallel version of the function. You may use the constructs `task` and/or `parallel`, as seen in the lectures.

Question 2

Imagine that the data structure you are given, instead of an `Array[A]`, is one called `ParSeq[A]`. This class offers the two following methods, which work in parallel:

```
def map[B](f: A => B): ParSeq[B]
def reduce(f: (A, A) => A): A
```

Can you write the following `minMax` function in terms of `map` and/or `reduce` operations ?

```
def minMax(data: ParSeq[Int]): (Int, Int) = ???
```

Question 3

What property does the function `f` passed to `reduce` need to satisfy in order to have the same result regardless on how `reduce` groups the applications of the operation f to the elements of the data structure? Prove that your function `f` indeed satisfies that property.