Lisbon Institute of Engineering

Degree in Informatics and Computer Engineering

Concurrent Programming

Summer 2022/2023, Second Series of Financial Years

Solve the following exercises and submit the tests with which you validated the correctness of the implementation of each exercise. The submission must be made by creating the **0.2.0** tag in each student's individual repository.

1. Implement a class with the same functionality as the **java.util.concurrent.CyclicBarrier** class.

2. Implement, without using *locks*, a *thread-safe* version of the **UnsafeContainer** class that stores a set of values and the number of times these values can be consumed.

```
class UnsafeValue<T>(val value: T, var initialLives: Int)
class UnsafeContainer<T>(private val values: Array<UnsafeValue<T>>){
    private var index = 0
    fun consume(): T? {
        while(index < values.size) {
            if (values[index].lives > 0) {
                values[index].lives -= 1
                return values[index].value
            }
            index += 1
        }
        return null
    }
}
```

As an example, the container built by **Container(Value("isel", 3), Value("pc", 4))** returns, via the **consume** method, the string **"isel"** three times and the string **"pc"** four times. After that, all calls to **consume** return **null**.

3. Consider the following *non-thread-safe* implementation of an object container with a usage count, which automatically calls the **close** function when the usage count is zero. Create a *thread-safe* version of this class without using *locks.*

```
class UnsafeUsageCountedHolder<T : Closeable>(value: T) { private
    var value: T? = value
    // the instance creation counts as one usage
    private var useCounter: Int = 1

    fun tryStartUse(): T? {
        if (value == null) return null
        useCounter += 1
        return value
    }

    fun endUse() {
        if (useCounter == 0) throw IllegalStateException("Already closed") if
        (--useCounter == 0) {
            value?.close()
            value = null
        }
    }
}
```

4. Implement the function **fun <T> any(futures: List<CompletableFuture<T>>): CompletableFuture<T>** which, given a non-empty list of *futures*, returns a complete *future*:
   - Successfully, when any *future* in the list is successfully completed. The value of the *future* returned must be the value of the *future* in the list that was completed.
   - With the exception of when all the *futures* on the list are completed. With the exception of the *future* returned should aggregate the exceptions of all the *futures* in the list.

This functionality is similar to that of the **Promise.any** function in the JavaScript language. Minimizing the acquisition of *locks is* valued in the implementation of this function.

Deadline for submission: May 14, 2023

ISEL, April 17th, 2023