Lisbon Institute of Engineering

Degree in Informatics and Computer Engineering
Concurrent Programming
Summer 2022/2023, First Series of Financial Years

Make *thread-safe* classes with the implementation of the following synchronizers. For each synchronizer, present at least one of the programs or tests you used to verify the correctness of the respective implementation. The resolution should also contain documentation, in the form of comments in the source file, including:

- Technique used (e.g. *monitor-style* vs. *execution/kernel-style* delegation).
- Non-obvious implementation aspects.

Delivery must be done by creating the **0.1.0** *tag* in each student's individual repository.

1. Implement the **NAryExchanger** synchronizer with functionality similar to the **Exchanger** synchronizer **of the** same name present in the standard Java library, but generalized for dimensions greater than 2.

```
class NAryExchanger<T>(groupSize: Int) {
    @Throws(InterruptedException::class)
    fun exchange(value: T, timeout: Duration): List<T>? { ... }
}
```

This synchronizer supports the exchange of information between groups of *threads* with **groupSize**, where **groupSize** is greater than or equal to two. The *threads* that use this synchronizer express their willingness to start an **exchange** by invoking the exchange method, specifying the object they want to deliver to the group (**value**) and the **time** limit for waiting for the exchange to take place (**timeout**). The **exchange** method ends by: (a) returning the values made available by all the threads in the group; (b) returning **null**, if the specified timeout limit expires, or; (c) throwing an **InterruptedException** when the *thread*'s wait is interrupted.

A group is formed and the exchange is performed whenever there are **groupSize** calls on hold. In addition:

a) If the call *C* returned the list *L*, then the size of *L* is **groupSize**.
b) If the *C* call returned the *L* list, then the value delivered in the *C* call is present in the *L* list.
c) If the call C_1 returned the list *L*, containing the values delivered by the calls C_1 , ... C_N , then the call C_i also returned the list *L*, for $1 \leq i \leq N$.

2. Implement the *blocking message queue* synchronizer, to support communication between producing and consuming *threads* via messages of the generic type **T.** Communication should use the FIFO (*first in first out*) criterion: given two messages placed in the queue, the first to be delivered to a consumer should be the first one delivered to the queue; if there are two or more consumers waiting for a message, the first to have its request satisfied is the one that has been waiting the longest. The maximum number of elements stored in the queue is determined by the **capacity** parameter, defined in the constructor.

   The public interface of this synchronizer is as follows:

```
class BlockingMessageQueue<T>(private val capacity: Int) {
    @Throws(InterruptedException::class)
    fun tryEnqueue(message: T, timeout: Duration): Boolean { ... }
    @Throws(InterruptedException::class)
    fun tryDequeue(nOfMessages: Int, timeout: Duration): List<T>? { ... }
}
```

   The **tryEnqueue** method delivers a message to the queue and is blocked if the queue has no available capacity for that message. This block should end when the message can be placed on the queue without exceeding its capacity, or delivered to a consumer. If the defined time is exceeded, the method should return **false**.

   The **tryDequeue** method removes and returns a list of messages from the queue, with dimension **nOfMessages**. blocked for as long as the request cannot be completely fulfilled. The block is limited by the duration defined by **timeout**. If this time is exceeded, the method must return **null**.

   Both methods must be sensitive to interruptions, handling them according to the protocol defined in the Java platform.

3. Implement the *thread pool executor* synchronizer, in which each command submitted is executed in one of the *worker threads* that the synchronizer creates and manages for this purpose. The public interface of this synchronizer is as follows:

```
class ThreadPoolExecutor(
    private val maxThreadPoolSize: Int,
    private val keepAliveTime: Duration,
) {
    @Throws(RejectedExecutionException::class)
    fun execute(runnable: Runnable): Unit { ... }
    fun shutdown(): Unit { ... }
    @Throws(InterruptedException::class)
    fun awaitTermination(timeout: Duration): Boolean { ... }
}
```

The maximum number of *worker threads* (**maxThreadPoolSize**) and the maximum time a *worker thread* can be inactive before terminating (**keepAliveTime**) are passed as arguments to the constructor of the **ThreadPoolExecutor** class.

The management of *worker threads by the synchronizer* must comply with the following criteria: (1) if the total number of *worker threads* is less than the maximum limit specified, a new *worker thread is* created whenever a *runnable is* submitted for execution and there is no *worker thread* available; (2) the *worker threads* must end after the time specified in **keepAliveTime has** elapsed without being mobilized to execute a command; (3) the number of *worker threads in* the *pool* at any given time depends on its activity and can vary between zero and **maxThreadPoolSize**.

Threads that want to execute functions via the *executor thread pool* invoke the **execute** method, specifying the command to execute with the **runnable** argument. This method returns immediately.

Calling the **shutdown** method puts the executor into shutdown mode and returns immediately. In this mode, all calls to the **execute** method should throw the **RejectedExecutionException** exception. However, all execution submissions made before the call to the **shutdown** method should be processed normally.

The **awaitTermination** method allows any invoking *thread to* synchronize with the completion of the executor's shutdown process, i.e. it waits until all accepted commands have been executed and all active *worker threads have terminated,* and it can terminate: (a) normally, by returning **true**, when the executor *shutdown* is complete; (b) exceptionally, by returning **false**, if the **time** limit specified with the **timeout** argument expires, without the shutdown ending, or; (c) exceptionally, by throwing **InterruptedException**, if the *thread*'s blocking is interrupted.

4. Add the following method to the *thread pool executor* made in question 3.

```
fun <T> execute(callable: Callable<T>): Future<T> { ... }
```

which schedules the execution of a *callable* and immediately returns a *representative of* that operation, implementing the **Future<T>** and *thread-safe* interface. To solve this exercise, do not use **Future<T>** implementations that exist in the Java platform's class library. All potentially blocking methods of **Future<T>** implementations must be sensitive to interruptions, handling them according to the protocol defined in the Java platform.

Deadline for submission: April 16, 2023

ISEL, March 20, 2023