Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente
Teste Global da Época Especial, **Verão** de 2021/2022
Duração: 3 horas

---

1. [5] Implemente o sincronizador **`Exchanger`** com funcionalidade semelhante ao sincronizador com o mesmo nome presente na biblioteca standard do Java.

```
class Exchanger<T> {
    @Throws(InterruptedException::class)
    fun exchange(value: T, timeout: Duration): Pair<Thread, T>? { ... }
}
```

Este sincronizador suporta a troca de informação entre pares de *threads*. As *threads* que utilizam este sincronizador manifestam a sua disponibilidade para iniciar uma troca invocando o método **`exchange`**, especificando o objecto que pretendem entregar à *thread* parceira (**`value`**) e a duração limite da espera pela realização da troca (**`timeout`**). O método **`exchange`** termina: (a) devolvendo o valor trocado e a *thread* que o forneceu, quando é realizada a troca com outra *thread*; (b) devolvendo **`null`**, se expirar o limite do tempo de espera especificado, ou; (c) lançando **`InterruptedException`** quando a espera da *thread* for interrompida.
Na implementação garanta que se a chamada C1 retornou o valor submetido pela chamada C2, então a chamada C2 retornou obrigatoriamente o valor submetido pela chamada C1.

2. [5] Implemente o sincronizador *message queue*, para suportar a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico **`T`**. A comunicação deve usar o critério FIFO (*first in first out*). A interface pública deste sincronizador é a seguinte:

```
class MessageQueue<T>() {
    @Throws(InterruptedException::class)
    fun tryEnqueue(message: T, timeout: Duration): Thread? { ... }
    @Throws(InterruptedException::class)
    fun tryDequeue(timeout: Duration): T? { ... }
}
```

O método **`tryEnqueue`** entrega uma mensagem à fila, retornando a referência para a *thread* que consumiu a mensagem. O método **`tryEnqueue`** fica bloqueado até que: 1) a mensagem entregue seja entregue a um consumidor, 2) o tempo **`timeout`** definido para a operação não expirar, ou 3) a *thread* não for interrompida. O método **`tryDequeue`** tenta remover uma mensagem da fila, bloqueando a *thread* invocante enquanto: essa operação não puder ser concluída com sucesso, 2) o tempo **`timeout`** definido para a operação não expirar, ou 3) a *thread* não for interrompida. Deve ser garantida ordem FIFO (*first in first out*) em todas as operações.

3. [3] Implemente a seguinte classe *thread-safe*, sem recurso à utilização de *locks*.

```
class CounterModulo(moduloValue: Int) {
    val value: Int
    fun increment(): Int
    fun decrement(): Int
}
```

Este classe implementa um contador, cujo valor pode estar entre 0 e **moduloValue** (exclusive). O método **increment** incrementa o valor do contador e caso este seja **moduloValue-1**, então o resultado do incremento deve ser **0**. O método **decrement** decrementa o valor do contador e caso este seja **0**, então o resultado do decremento deve ser **moduloValue-1**. Ambos os métodos retornam o resultado da operação.

4. [3] Realize o sincronizador **MessageBox** com a interface apresentada em seguida.

```
class MessageBox<T> {
    suspend fun waitForMessage(): T { ... }
    fun sendToOne(message: T): Int { ... }
}
```

A função **waitForMessage** suspende a corrotina onde foi realizada a invocação até que uma mensagem seja enviada através da função **sendToOne**. A função **sendToOne** deve retornar o número exacto de chamadas a **waitForMessage** que receberam a mensagem, podendo este valor ser zero (não existiam corrotinas à espera de mensagem), ou um. A mensagem passada na chamada **sendToOne** não pode ser entregue a mais do que um consumidor e não pode ficar disponível para chamadas futuras da função **waitForMessage**.

5. [4] Implemente a função com a seguinte assinatura

```
suspend fun <A,B,C> run(f0: suspend ()->A, f1: suspend ()->B, f2: suspend (A,B)->C): C
```

que retorna o valor da expressão **f2(f1(), f0())**, realizando a computação de **f0()** e **f1()** em paralelo.

Duração: 3 horas
ISEL, 16 de setembro de 2022