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

---

1. [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>() {
   fun enqueue(message: T): Unit { … }
   @Throws(InterruptedException::class)
   fun tryDequeue(nOfMessages: Int, timeout: Duration): List<T>? { … }
}
```

   O método **`enqueue`** entrega uma mensagem à fila nunca ficando bloqueado. O método **`tryDequeue`** tenta remover **`nOfMessages`** mensagens da fila, bloqueando a *thread* invocante enquanto: essa operação não puder ser concluída com sucesso, ou 2) o tempo **`timeout`** definido para a operação não expirar, ou 3) a *thread* não for interrompida. A remoção não pode ser realizada parcialmente, i.e., ou **`nOfMessages`** mensagens são removidas ou nenhuma mensagem é removida. Estas operações de remoção devem ser completadas pela ordem de chegada, independentemente dos valores de **`nOfMessages`**. Tenha em atenção as consequências de uma desistência, por cancelamento ou *timeout*, de uma operação de **`tryDequeue`**.

2. [5] Implemente o sincronizador semáforo com capacidade de shutdown, com a seguinte interface pública:

```
class Semaphore(private val initialUnits: Int) {
   fun release(): Unit { … }
   @Throws(InterruptedException::class, RejectedExecutionException::class)
   fun acquire(timeout: Duration): Boolean { … }
   fun shutdown(): Unit { … }
   @Throws(InterruptedException::class)
   fun awaitTermination(timeout: Duration): Boolean { … }
}
```

   O método **`release`** entrega uma unidade ao semáforo, nunca ficando bloqueado. O método **`acquire`** tenta adquirir uma unidade, bloqueando a *thread* invocante enquanto: 1) essa operação não puder ser concluída com sucesso, ou 2) o tempo **`timeout`** definido para a operação não expirar, ou 3) a *thread* não for interrompida.
   O método **`shutdown`** coloca o sincronizador em modo de *shutting-down*, onde todas as operações de aquisição pendentes ou futuras terminam com a excepção **`RejectedExecutionException`**. O método **`awaitTermination`** espera até que: 1) o semáforo esteja em modo de *shutting down* e 2) que o número de unidades seja igual ao número inicial de unidades. Este método: 1) recebe o tempo máximo de espera, 2) deve também ser sensível a interrupções, 3) pode ser chamado por mais do que uma *thread*.

3. [3] Implemente, sem utilizar *locks*, uma versão *thread-safe* da classe **`UnsafeContainer`** que armazena um conjunto de valores e o número de vezes que esses valores podem ser consumidos.

```
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
    }
}
```

A título de exemplo, o contentor construído por **`Container(Value("isel", 3), Value("pc", 4))`** retorna, através do método **`consume`**, a string **`"isel"`** três vezes e a string **`"pc"`** quatro vezes. Depois disso, todas as chamadas a **consume** retornam **null**.

4. [3] Considere o sincronizador **`Exchanger`** realizado na primeira série de exercícios. Realize um sincronizador com funcionalidade semelhante, mas em que a função **`exchange`** é **`suspend`**, ou seja, não bloqueia a *thread* invocante durante a espera, e não suporta *timeout* nem cancelamento.

```
class Exchanger<T> {
    suspend fun exchange(value: T): T { … }
}
```

Este sincronizador suporta a troca de informação entre pares de corrotinas. As corrotinas que utilizam este sincronizador manifestam a sua disponibilidade para iniciar uma troca invocando o método **`exchange`**, especificando o objecto que pretendem entregar à corrotina parceira (**`value`**). O método **`exchange`** termina devolvendo o valor trocado, quando é realizada a troca com outra corrotina.

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

```
suspend fun race(f0: suspend () -> Int, f1: suspend () -> Int): Int
```

Esta função executa de forma paralela as funções passadas como argumento, retornando o valor retornado pela primeira função a acabar com sucesso. Uma execução da função **`race`** só deve acabar quando as corrotinas criadas no seu âmbito tiverem terminado. Contudo, a função **`race`** deve terminar o mais depressa possível, através do cancelamento da corrotina ainda em execução, após uma das funções passadas como argumento tiver terminado com sucesso.

Duração: 3 horas
ISEL, 6 de julho de 2022