Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente
Teste Global da Época Normal, Inverno de 2021/2022
Duração: 2 horas

---

1. [5] Implemente na linguagem *Java* o sincronizador *message box*, com os métodos apresentados em seguida.

```
public class MessageBox<T> {
    public Optional<T> waitForMessage(long timeout) throws InterruptedException;
    public int sendToMany(T message, int min, int max);
}
```

O método **waitForMessage** bloqueia a *thread* invocante até que uma mensagem seja enviada através do método **sendToMany**. O método **waitForMessage** pode terminar com: 1) um objecto **Optional** contendo a mensagem enviada; 2) um objecto **Optional** vazio, caso o tempo de espera definido por **timeout** seja excedido sem que uma mensagem seja enviada; 3) com o lançamento duma excepção do tipo **InterruptedException**, caso a *thread* seja interrompida enquanto em espera.

O método **sendToMany** envia uma mensagem para o máximo de **max** consumidores, se existirem pelo menos **min** consumidores. Deve retornar o número exacto de *threads* que receberam a mensagem, podendo este valor ser zero (não existiam *threads* suficientes à espera de mensagem), ou estar entre **min** e **max** inclusive. A mensagem passada na chamada **sendToMany** não deve ficar disponível para chamadas futuras do método **waitForMessage**.

2. [5] Implemente na linguagem *Java* a variante do sincronizador *exchanger* com a interface apresentada em seguida.

```
public class Exchanger2<A, B> {
    public Optional<B> exchangeA(A elem, long timeout) throws InterruptedException;
    public Optional<A> exchangeB(B elem, long timeout) throws InterruptedException;
}
```

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 **exchangeA** ou **exchangeB**, especificando o objecto que pretendem entregar à *thread* parceira (**elem**) e, opcionalmente, o tempo limite da espera pela realização da troca (**timeout**). A chamada ao método **exchangeA** ou **exchangeB** termina: (a) devolvendo um *optional* com valor, quando é realizada a troca com outra *thread*, sendo o objecto por ela oferecido retornado no valor desse *optional*; (b) devolvendo um *optional* vazio, se expirar o limite do tempo de espera especificado, ou; (c) lançando **InterruptedException** quando a espera da *thread* for interrompida.

Note que existem dois tipos de mensagens, em vez de apenas um tipo como no sincronizador *exchanger* presente na biblioteca *standard* do Java. Note também que se a chamada **A1** do método **exchangeA** retornar o elemento passado na chamada **B1** do método **exchangeB**, então a chamada **B1** retorna obrigatoriamente o elemento passado na chamada **A1**.

3. [5] Realize na linguagem C# a classe com a interface apresentada em seguida.

```
public class CountDownLatchWithAutoReset {
    public CountDownLatchWithAutoReset(int initialValue);
    public void Decrement();
    public Task WaitForZeroAsync();
}
```

Após construção, cada instância fica com o valor definido por **initialValue**. O método `WaitForZeroAsync` retorna uma *task* que ficará completa quando o valor do contador chegar a zero. O método `Decrement` decrementa o valor do contador. Caso este valor chegue a zero, o contador é reiniciado com o valor **initialValue**, garantindo contudo que todas as *tasks* associadas a chamadas anteriores de **WaitForZeroAsync** são completadas.

4. [2] Realize na linguagem C# a função

```
Task<T[]> WhenAll<T>(
    Func<CancellationToken, Task<T>> op1, Func<CancellationToken, Task<T>> op2,
    CancellationToken ct);
```

Esta função deve iniciar as operações canceláveis representadas por **op1** e **op2** de forma paralela, tirando partido da interface programática para essas operações ser assíncrona. A função **WhenAll** deve retornar uma *task* que se completa com sucesso apenas quando ambas as operações tiverem terminado, usando a mesma caracterização funcional da função **Task.WhenAll**. Em adição, deve ser solicitado cancelamento das operações representadas por **op1** e **op2** caso: a) o *token* **ct** solicite cancelamento; b) qualquer uma das operações termine sem sucesso (e.g. se a primeira operação terminar com erro deve ser solicitado imediatamente cancelamento da segunda operação). Pode implementar métodos auxiliares, caso ache necessário, e pode também usar o método **Task.WhenAll**.

5. [3] Implemente na linguagem *Java* a seguinte classe *thread-safe*, sem recurso à utilização de *locks*.

```
public class Counter {
    public Counter(int maxValue);
    public int getValue();
    public Optional<Integer> increment();
    public int resetToZero();
}
```

Este classe implementa um contador, cujo valor pode estar presente entre 0 e **maxValue** (inclusive). O método **add** tenta incrementar o valor do contador, retornando o novo valor resultante do incremento ou *empty* caso o incremento não seja possível. O método **resetToZero** coloca o contador com o valor zero, retornando o valor anterior de contagem.

Duração: 2 horas
ISEL, 31 de janeiro de 2022