Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente
Teste Global da Época de Recurso, Verão de 2020/2021
Duração: 2 horas

---

1. [6] Implemente usando a linguagem Java o sincronizador *batch exchanger*, com os métodos apresentados em seguida.

```
public class BatchExchanger<T> {
   public BatchExchanger(int batchSize);
   public Optional<List<T>> deliverAndWait(T msg, long timeout) throws InterruptedException;
}
```

O método **`deliverAndWait`** entrega uma mensagem ao sincronizador e espera até que estejam presentes **`batchSize`** mensagens, i.e., até que um lote de mensagens esteja completo, retornando todas as mensagens deste lote. Uma chamada a **`deliverAndWait`** pode terminar com: 1) retorno dum objecto **`Optional`** contendo uma lista com todas as mensagens do lote; 2) um objecto **`Optional`** vazio, caso o tempo de espera definido por **`timeout`** seja excedido sem que o lote esteja completo; 3) com o lançamento duma excepção do tipo **`InterruptedException`**, caso a *thread* seja interrompida enquanto em espera. Assumindo que as chamadas a **`deliverAndWait`** entregam mensagens distintas**,** o sincronizador deve garantir as seguintes propriedades: 1) caso a chamada ao método **`deliverAndWait`** acabe sem sucesso, então a mensagem entregue nesta chamada não pode ser retornada em nenhuma outra chamada a esse método; 2) caso a chamada ao método **`deliverAndWait`** acabe com sucesso, então a lista retornada contém a mensagem entregue; 3) sejam **C1** e **C2** quaisquer duas chamadas ao método **`deliverAndWait`** que terminaram com sucesso, se o retorno de **C1** contém a mensagem entregue por **C2** então o retorno de **C2** contém a mensagem entregue por **C1**.

2. [6] Realize usando a linguagem Java o sincronizador **`NAryMessageQueue`** com a interface apresentada em seguida.

```
public class NAryMessageQueue<T> {
   public NAryMessageQueue<T>()
   public void put(T msg);
   public Optional<List<T>> get(int n, long timeout) throws InterruptedException;
}
```

Este sincronizador implementa uma fila de mensagens em que a inserção é unária e não bloqueante; e a remoção é n-ária e potencialmente bloqueante. A atribuição das mensagens aos pedidos **`get`** segue a ordem FIFO (*first in first out*): se o pedido **G2** foi realizado depois do pedido **G1**, então o pedido **G2** só pode ser concluído com sucesso depois do pedido **G1** ser concluído (com sucesso ou insucesso). Uma chamada a **`get`** pode terminar com: 1) retorno dum objecto **`Optional`** contendo a lista com **`n`** mensagens; 2) um objecto **`Optional`** vazio, caso o tempo de espera definido por **`timeout`** seja excedido sem que o pedido seja concluído; 3) com o lançamento duma excepção do tipo **`InterruptedException`**, caso a *thread* seja interrompida enquanto em espera. Minimize o número de comutações de contexto.

3. [5] Considere a classe **`ManualResetEvent`** com os métodos apresentados em seguida. Cada instância pode estar num de dois estados: *set* e *clear*. Uma chamada ao método **`Set`** coloca a instância no estado *set*; uma chamada ao método **`Clear`** coloca a instância no estado *clear*. Uma chamada ao método **`WaitAsync`** retorna uma instância de **`Task`** que irá estar completa quando o estado for *set*, o que pode acontecer imediatamente ou algures no futuro.

```
public class ManualResetEvent
{
   public Task WaitAsync();
   public void Set();
   public void Clear();
}
```

   a. Implemente a classe **`ManualResetEvent`** com a interface apresentada.
   b. Implemente a variante da classe **`ManualResetEvent`** em que o método **`WaitAsync`** é cancelável através de um *cancellation token* passado como parâmetro.

4. [3] Considere o seguinte método:

```
public static T[] Oper<T>(T[] xs, T[] ys)
{
   if(xs.Length != ys.Length) throw new ArgumentException("lengths must be equal");
   var res = new T[xs.Length];
   for (var i = 0; i < xs.Length; ++i)
   {
       res[i] = B(A(xs[i]), A(ys[i]));
   }

   return res;
}
```

Realize na linguagem C# a versão assíncrona do método **`Oper`**, seguindo o padrão TAP (*Task-based Asynchronous Pattern*). Assuma que tem à disposição versões assíncronas dos métodos **`A`** e **`B`**, que não produzem efeitos colaterais e podem ser chamados em concorrência. Tire partido do paralelismo potencial existente. Não é necessário suporte para cancelamento.

Duração: 2 horas
ISEL, 27 de julho de 2021