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

---

1. [6] Realize o sincronizador **`SemaphoreWithWaitFull`**, que representa um semáforo com aquisição e libertação unária, sem garantia de ordem na atribuição de unidades e com a interface apresentada em seguida

```
public class SemaphoreWithWaitFull<E> {
  public SemaphoreWithWaitFull(int initialUnits);
  public boolean acquireSingle(long timeout) throws InterruptedException;
  public void releaseSingle();
  public boolean waitFull(long timeout) throws InterruptedException;
}
```

   O método **`waitFull`** deve ficar bloqueado enquanto o número de unidades disponíveis for menor que o valor inicial ou existirem consumidores em espera (i.e. chamadas a **`acquireSingle`** pendentes). Isto é, sempre que o semáforo passe por um estado em que o número de unidades for igual ao inicial e não existir nenhum consumidor em espera, deve ser realizado o desbloqueio e retorno com sucesso de todas as chamadas de **`waitFull`** pendentes nesse momento.
   Os restantes métodos, **`acquireSingle`** e **`releaseSingle`**, tem o comportamento típico de um semáforo unário sem garantia de ordem.

2. [6] Realize o sincronizador *broadcast box* cuja interface pública, em *Java*, é a seguinte:

```
public class BroadcastBox<E> {
  public int deliverToAll(E message, long timeout) throws InterruptedException;
  public Optional<E> receive(long timeout) throws InterruptedException;
}
```

   O método **`deliverToAll`** entrega uma mensagem a *todos* os consumidores à espera de receber mensagem nesse momento. Caso não exista nenhum consumidor, aguarda no máximo **`timeout`** milissegundos até que exista pelo menos um consumidor presente. Este método retorna o número exacto de consumidores que receberam a mensagem, que pode ser zero em caso de *timeout*, ou lança **`InterruptedException`** se a espera da *thread* for interrompida. O método **`receive`** permite receber uma mensagem, e termina: (a) com sucesso, retornado um **`Optional`** com a mensagem recebida; (b) retornando **`Optional.empty()`** se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`InterruptedException`** quando a espera da *thread* é interrompida.

3. [3] Considere o seguinte método:

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

   return acc;
}
```

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 chamadas em concorrência. Assuma também que tem à disposição a versão assíncrona do método **`C`**, e que esta não é associativa. Tire partido do paralelismo potencial existente. Não é necessário suporte para cancelamento.

4. [5] Considere a classe **`CountdownLatch`** com os métodos apresentados em seguida. Cada instância possui um valor de contagem que apenas pode ser decrementado, através do método **`Decrement`**. Uma chamada ao método **`WaitAsync`** retorna uma **`Task`** que é completada com sucesso quando o valor de contagem for zero.

```
public class CountdownLatch
{
   public CountdownLatch(init initialCount)
   public Task WaitAsync();
   public void Decrement();
}
```

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

Duração: 2 horas
ISEL, 17 de setembro de 2021