Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente
Teste Global da Época de Recurso, Inverno de 2020/2021

---

1. [2] Considere a classe **`UnsafeSpinLifoMsgQueue`** apresentada em seguida, com uma implementação não *thread-safe* de uma fila de mensagens com ordem LIFO (*last-in-first-out*) e espera activa. Sem usar *locks*, implemente uma versão *thread-safe* desta classe.

```
public class UnsafeSpinLifoMsgQueue<T> {

   private static class Node<E> {
       Node<E> next;
       E msg;

       Node(E msg) {
           this.msg = msg;
       }
   }

   private Node<T> first;

   public void Put(T msg) {
       Node<T> node = new Node<T>(msg);
       node.next = first;
       first = node;
   }

   public T Take() {
       Node<T> oldFirst;
       while ((oldFirst = first) == null) {
           Thread.yield();
       }
       first = oldFirst.next;
       return oldFirst.msg;
   }
}
```

2. [5] Realize o sincronizador **`SemaphoreWithShutdown`**, 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 SemaphoreWithShutdown<E> {
  public SemaphoreWithShutdown(int initialUnits);
  public boolean acquireSingle(long timeout)
                                  throws InterruptedException, CancellationException;
  public void releaseSingle();
  public void startShutdown();
  public boolean waitShutdownCompleted(long timeout) throws InterruptedException;
}
```

O método **`startShutdown`** coloca o semáforo num estado de encerramento. Nesse estado todas as chamadas a **`acquireSingle`**, futuras ou atualmente pendentes, devem terminar com o lançamento da excepção **`CancellationException`**. O processo de encerramento é considerado completo quando as unidades

disponíveis no semáforo forem iguais ao valor inicial, definido na construção. Chamadas ao método **`waitShutdownCompleted`** esperam que o processo de encerramento esteja concluído.
Os métodos **`acquireSingle`** e **`waitShutdownCompleted`** recebem o valor do tempo máximo de espera, retornando **`false`** se e só o fim da sua execução se dever à expiração desse tempo. Ambos os métodos devem ser sensíveis a interrupções, tratando-as de acordo com o protocolo do Java para métodos potencialmente bloqueantes. Minimize o número de objetos alocados durante a operação do semáforo, bem como as comutações de contexto.

3. [6] Realize o sincronizador **`PairTransferQueue`** com a interface apresentada em seguida

```
public class PairTransferQueue<A, B> {
  public boolean transfer0(A message, long timeout) throws InterruptedException;
  public boolean transfer1(B message, long timeout) throws InterruptedException;
  public Optional<Pair<A,B>> take(long timeout) throws InterruptedException;
}
```

Este sincronizador implementa uma fila de transferência de mensagens, semelhante à realizada na primeira série de exercícios, com as seguintes diferenças.
Existem dois métodos de transferência: **`transfer0`** e **`transfer1`**, que transferem mensagens de tipos eventualmente diferentes. Esses métodos são potencialmente bloqueantes, retornando **`true`** apenas quando a mensagem entregue tiver sido retirada via o método **`take`**. Os métodos **`transfer0/1`** podem também retornar **`false`**, no caso do tempo de espera ter expirado, ou acabar com o lançamento da excepção **`InterruptedException`** em caso de interrupção. Nestes dois casos, expiração de tempo e interrupção, deve ser garantido que a mensagem não foi removida por um **`take`**.
O método **`take`** é potencialmente bloqueante, retornando um **`Optional`** com um par de mensagens, uma de cada tipo, ou **`Optional.empty()`** caso não seja possível remover o par dentro do tempo definido. O método **`take`** é também sensível a interrupções.
O sincronizador deve usar um critério FIFO (*first in first out*) para a finalização com sucesso das operações **`transfer0`**, **`transfer1`** e **`take`**. Por exemplo, uma chamada a **`transfer0`** só deve ser concluída com sucesso quando todas as chamadas a **`transfer0`** anteriores tenham sido concluídas.

4. [3] Considere o seguinte método:

```
public T[] Compute<T>(T[] xs)
{
   var res = new T[xs.Length];
   for (var i = 0; i < xs.Length; ++i)
   {
       res[i] = Oper(Oper(xs[i]));
   }
   return res;
}
```

Realize na linguagem C# a versão assíncrona do método **`Compute`**, seguindo o padrão TAP (*Task-based Asynchronous Pattern*). Assuma que tem à disposição uma versão assíncrona do método **`Oper`**. A operação associada a este método não produz efeitos colaterais, podendo existir várias execuções da operação em simultâneo. Contudo, a operação associada ao método **`Oper`** pode produzir excepções, devendo a execução da operação ser retentada após um período de espera de 100 milissegundos. O número máximo total de retentativas é definido pelo parâmetro **`maxRetries`** do método **`Compute`**. Quando este número é alcançado, a execução da operação associada a **`Compute`** deve terminar com excepção.
Tire partido do paralelismo potencial existente.

5. [4] Realize em C# a class **`Exchanger`**

```
public class Exchanger<T>
{
   public Task<T> ExchangeAsync(T message);
}
```

Este tipo suporta a troca assíncrona de mensagens entre pares de chamadas a **`ExchangeAsync`**: se a *task* retornada pela chamada **C1** se completar com a mensagem fornecida pela chamada **C2**, então a *task* retornada na chamada **C2** completa-se com a mensagem fornecida pela chamada **C1**. As chamadas ao método **`ExchangeAsync`** devem retornar o mais depressa possível, com uma *task* completada ou não completada.

a. Implemente a versão da classe **`Exchanger`** sem suporte a cancelamento no método **`ExchangeAsync`**.

b. Implemente a versão da classe **`Exchanger`** com suporte a cancelamento no método **`ExchangeAsync`**, que passa a receber também uma instância de **`CancellationToken`**. Note que se a *task* retornada pela chamada C1 se completar com cancelamento, então a mensagem fornecida pela chamada C1 não pode ser usada para completar com sucesso qualquer *task* retornada por outra chamada.

Duração: 2 horas e 30 minutos
ISEL, 18 de fevereiro de 2021