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

---

1. [2] Considere a classe **`UnsafeMessageBox`**, cuja implementação em C# se apresenta a seguir:

```
public class UnsafeMessageBox<M> where M : class {
  private class MsgHolder {
    internal readonly M msg;
    internal int lives;
  }
  private MsgHolder msgHolder = null;
  public void Publish(M m, int lvs) {msgHolder = new MsgHolder { msg = m, lives = lvs };}
  public M TryConsume() {
    if (msgHolder != null && msgHolder.lives > 0) {
      msgHolder.lives -= 1;
      return msgHolder.msg;
    }
    return null;
  }
}
```

   Esta implementação reflete a semântica de um sincronizador *message box* contendo no máximo uma mensagem que pode ser consumida múltiplas vezes, até ao máximo de **`lvs`**. Contudo esta classe não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, uma versão *thread-safe* deste sincronizador.

2. [5] Implemente em *Java*, com base nos monitores implícitos ou explícitos, o sincronizador *broadcast box* cuja interface pública, em *Java*, é a seguinte:

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

   O método **`deliverToAll`** entrega uma mensagem a todas as *threads* à espera de receber mensagem nesse momento e nunca bloqueia a *thread* invocante. Este método retorna o número exacto de *threads* que receberam a mensagem; se não existem *threads* bloqueadas a mensagem é descartada e o método retorna 0. 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. [6] Realize o sincronizador **`TransferQueueWithShutdown`**, que representa uma fila de mensagens com garantia de ordem, tanto na entrega como na recepção, semelhante à realizada na primeira série de exercícios. Tem a interface apresentada em seguida:

```
public class TransferqueueWithShutdown<E> {
  public TransferQueueWithShutdown();
  public void put(E message) throws CancellationException;
  public Optional<E> take(long timeout) throws InterruptedException,
                                               CancellationException;
  public void startShutdown();
  public boolean waitShutdownCompleted(long timeout) throws InterruptedException;
}
```

   O método **`put`** entrega uma mensagem à fila, **sem sincronização** com a recepção dessa mensagem. Pode lançar **`CancellationException`** caso a fila já esteja em processo de *shutdown*.

O método **take** retira uma mensagem da fila, pelo que pode bloquear a *thread* invocante até que exista uma mensagem disponível, e termina: (a) com sucesso, retornando um **Optional** com a mensagem; (b) retornando um **Optional** vazio se for excedido o limite especificado para o tempo de espera; (c) lançando **InterruptedException** quando a espera da *thread* é Interrompida; e, (d) lançando **CancellationException** caso a fila entre ou esteja em *shutdown* e não existam mais mensagens disponíveis.
O método **startShutdown** coloca a fila num estado de encerramento. Nesse estado não deve aceitar mais entregas de mensagens e apenas deve aceitar as remoções necessárias para remover as mensagens já presentes na fila. O processo de encerramento é considerado completo quando não existirem mais mensagens presentes na fila. Chamadas ao método **waitShutdownCompleted** esperam que o processo de encerramento esteja concluído. Este método recebe o tempo máximo de espera e deve reagir adequadamente a interrupções da *thread* que realizou a chamada.

4. [3] Considere o seguinte método:

```
public T Compute<T>(T[] elems, T initial)  {
  T acc = initial;
  foreach (var elem in elems) {
    var aux = A(elem);
    acc = E(D(B(elem), C(elem)), acc);
  }
  return acc;
}
```

Os métodos **A**, **B**, **C**, e **D** são funções sem efeitos colaterais e passíveis de múltiplas execuções em paralelo. Todos esses métodos recebem **T** e retornam **T**. O método **E** realiza uma operação não associativa. Realize uma versão assíncrona do método **Compute** seguindo o padrão TAP (*Task-based Asynchronous Pattern*) usando os métodos assíncronos do C# e/ou a funcionalidades disponíveis na TPL. Assuma que tem disponível as versões TAP dos métodos **A**, **B**, **C**, **D**, e **E**. Tire partido do paralelismo potencial existente.

5. [4] Realize em C# a classe **CountdownLatch**, cuja interface se apresenta em seguida.

```
public class CountdownLatch
{
   // initializes counter with initialValue
   public CountdownLatch(int initialValue);

   // Decrements counter
   public void Countdown();

   // Waits for counter to be zero
   public Task WaitZeroAsync();

   // Waits for counter to be zero, supporting cancellation
   public Task WaitZeroAsync(CancellationToken ct);
}
```

Duração: 2 horas e 30 minutos
ISEL, 11 de março de 2021