# Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

**Programação Concorrente, Inverno de 2013/2014**

Teste Final (2ª Época)

1. [3,5] Considere a classe UnsafeSpinReadWriteLock, não *thread-safe*, apresentada a seguir em C#:

```
public class UnsafeSpinReadWriteLock {
   private int state;   // 0 means free, -1 means writing, and > 0 means reading
   public void LockRead() {
      SpinWait sw = new SpinWait();
      while (state < 0)
         sw.SpinOnce();
      state++;
   }
   public void LockWrite() {
      SpinWait sw = new SpinWait();
      while (state != 0)
         sw.SpinOnce();
      state = -1;
   }
   public void UnlockRead() { state--; }
   public void UnlockWrite() { state = 0; }
}
```

   Indique as razões pelas quais a implementação não é *thread-safe* e, sem recorrer à utilização de primitivas de sincronização bloqueantes, apresente as alterações necessárias (em Java ou C#) para a tornar *thread-safe*.

2. [4] Usando monitores implícitos do *Java* ou da CLI, implemente o sincronizador *blackboard*, que viabiliza a divulgação de mensagens com prazo de validade. A interface pública do sincronizador é apresentada de seguida em C#.

```
public class BlackBoard<T> where T : class {
   public void Write(T message, int duration);
   public T Read(int timeout);
   public void Clear();
}
```

   O sincronizador *blackboard* suporta a afixação de mensagens a serem lidas por várias *threads*. A operação Write afixa a mensagem recebida (message), que permanecerá válida durante o intervalo de tempo duration, especificado em milissegundos. No caso de ainda existir uma mensagem válida no *blackboard*, a nova mensagem substitui a existente.

   A operação Read promove a leitura da mensagem afixada no *blackboard*, bloqueando a *thread* invocante caso não exista nenhuma mensagem válida. A operação termina: com sucesso, retornando a última mensagem válida; ou com insucesso, lançando a respectiva excepção, caso o tempo máximo de espera (timeout) seja excedido ou o bloqueio d *thread* seja cancelado.

   A operação `Clear` remove, caso exista, a mensagem afixada no *blackboard*.

3. [5] Usando monitores intrínsecos da CLI, implemente o sincronizador *future*, cujas instâncias representam resultados de computações realizadas assincronamente. A interface pública do sincronizador é apresentada de seguida em C#.

```
public sealed class Future<T> : IAsyncResult {
   public Future(AsyncCallback userCallback, object asyncState);
   public T Result { get; }
   public bool TrySet(T result);
   public bool TrySetException(Exception exception);
```

```
    // IAsyncResult properties.
    public bool IsCompleted { get; }
    public WaitHandle AsyncWaitHandle { get; }
    public object AsyncState { get; }
    public bool CompletedSynchronously { get { return false; } }
  }
```

O resultado da computação realizada assincronamente é publicado através da operação TrySet, caso a computação tenha terminado normalmente, ou através da operação TrySetException, caso a computação tenha terminado por ter ocorrido uma condição excepcional. Uma vez publicado o resultado da computação, chamadas subsequentes a TrySet ou a TrySetException não produzem efeitos no estado do sincronizador, retornando false.

A propriedade Result produz o resultado da computação assíncrona, bloqueando a thread invocante até que o resultado da computação esteja disponível (i.e. seja publicado através das operações TrySet*). Sublinha-se que caso a computação assíncrona tenha terminado com erro, a propriedade Result lança a excepção produzida pela computação.

A sincronização com a conclusão da computação assíncrona é feita através das propriedades da interface IAsyncResult. A propriedade AsyncWaitHandle produz a referência para a instância de ManualResetEvent que fica sinalizado quando for publicado o resultado da operação assíncrona. Note que a instância de ManualResetEvent é criada de forma deferida caso seja efectivamente necessária, ou seja, se a propriedade AsyncWaitHandle for de facto acedida.

Para suportar o *rendezvous* com a conclusão da computação assíncrona usando c*allback*, o sincronizador fornece um construtor que recebe como argumento a instância de AsyncCallback que será invocada quando a computação for concluída. O outro parâmetro do construtor, asyncState, permite especificar um objecto que poderá ser obtido através da propriedade AsyncState.

4. [4] Implemente em C# a seguinte método estático, fazendo uso do suporte para I/O assíncrono baseado no APM:

```
public static WaitHandle ApmCopyStream (Stream src, Stream dst);
```

O método copia o conteúdo do *stream* de dados src para o *stream* dst e devolve um WaitHandle que representa o objecto de sincronização que será sinalizado quando a cópia termina.

A cópia deve ser feita em blocos de 4KiB, devendo a implementação assegurar paralelismo entre a escrita do bloco de ordem N e a leitura do bloco de ordem N + 1. Tenha em atenção que os dados têm que ser escritos no *stream* dst pela mesma ordem com que são lidos do *stream* src.

5. [3,5] Fazendo uso da TPL, realize o método estático da classe AsyncUtils

```
public static Task<int> MaxIndex(int[] data);
```

que produz assincronamente, e com o paralelismo possível, o índice do maior elemento do *array* recebido como parâmetro.

Sugestão: Considere uma implementação recursiva, onde o problema seja dividido ao meio em cada recursão.

Duração: 2 horas e 30 minutos

ISEL, 5 de Fevereiro de 2014