Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente, Verão de 2013/2014

Teste Final (1ª Época)

1. [3] Considere a classe UnsafeCountdownLatch, não *thread-safe*, cuja implementação em C# é apresentada a seguir:

```
public class UnsafeCountdownLatch {
   private int count;
   private readonly ManualResetEvent waitEvent = new ManualResetEvent(false);

   public UnsafeCountdownLatch(int c) {
      if (c <= 0) throw new InvalidOperationException();
      count = c;
   }

   public bool Wait(int timeout) {
      if (count == 0) return true;
      waitEvent.WaitOne(timeout);
      return count == 0;
   }

   public void Signal() {
      if (count == 0) throw new InvalidOperationException();
      if (--count == 0) waitEvent.Set();
   }

   public void AddCount() {
      if (count == 0 || count + 1 < 0) throw new InvalidOperationException();
      count++;
   }
}
```

   Indique as razões pelas quais esta classe não é *thread-safe* e, sem recorrer à utilização de primitivas de sincronização bloqueantes, apresente as alterações necessárias para a tornar *thread-safe*.

2. [4] Usando a linguagem *Java* e os respectivos monitores implícitos, implemente o sincronizador *transient signal* com a classe TransientSignal, cuja interface pública é a seguinte:

```
public class TransientSignal {
    public void await() throws InterruptedException;
    public void signalAll();
}
```

   A chamada ao método await bloqueia a *thread* invocante até que o método signalAll seja invocado, ou o bloqueio da *thread* seja interrompida. O método signalAll desbloqueia todas as *threads* em espera aquando da respectiva invocação, sendo ignorada a sinalização se não existirem *threads* bloqueadas. (O sincronizador deve respeitar a semântica de sincronização enunciada mesmo quando ocorrem notificações espúrias das *threads* bloqueadas no monitor implícito.)

3. [5] Considere a classe Combiner<L,R> que implementa o sincronizador *combiner* que agrega pares de objectos que lhe são entregues separadamente e entrega esses pares às *threads* que invocam o método Take. A interface pública em C# desta classe é a seguinte:

```
public class Pair<L,R> { public L left; public R right; }

public class Combiner<L,R> {
   public void PutLeft(L left);
   public void PutRight(R right);
   public Pair<L,R> Take(int timeout);
}
```

Cada par de objectos é constituído por um objecto esquerdo (left) e por um objecto direito (right), que são entregues separadamente ao sincronizador usando os métodos PutLeft e PutRight, respectivamente. O ritmo a que chegam os objectos esquerdos pode ser diferente do ritmo a que chegam os objectos direitos mas, sempre que for possível formar um par esquerdo/direito, esse par deve ser retornado como resultado da invocação do método Take. Implemente, em *Java* ou C#, a classe Combiner<L,R> garantindo que os objectos entregues com os métodos PutLeft e PutRight, bem como o atendimento das *threads* em espera é feita por ordem de chegada. O método Take pode terminar: com sucesso, retornando uma instância de Pair<L,R> contendo os dois objectos emparelhados; por esgotar o tempo máximo de espera (*timeout*), retornado null, ou; porque a espera da respectiva *thread* foi interrompida, lançando a respectiva excepção de interrupção.

4. [4] Implemente, na linguagem C#, a classe StreamFindBytesApm que determina, assincronamente, quantos *bytes* de um *stream* obedecem a um determinado predicado. A interface pública desta classe é a seguinte:

```
public class StreamFindBytesApm {
   public StreamFindBytesApm(Stream stream, Predicate<byte> filter);
   public long Occurrences { get; }
   public bool IsCompleted { get; }
   public WaitHandle AsyncWaitHandle { get; }
}
```

O processamento é iniciado na construção do objecto, contudo a *thread* que invoca o constructor não deve ficar bloqueada, aguardando que a contagem esteja terminada. Através da propriedade Occurrences é possível obter o número de *bytes* do *stream* que satisfazem o predicado, especificado através do argumento filter. A leitura desta propriedade deve bloquear a *thread* invocante até que a contagem esteja concluída. O evento subjacente ao WaitHandle que é retornado pela propriedade AsyncWaitHandle deve ficar no estado sinalizado e a propriedade IsCompleted deve devolver true quando o processamento do *stream* estiver concluído.

A implementação desta classe deve: ser baseada no modelo APM (*Asynchronous Programming Model*) do .NET *Framework*, processar os *bytes* do stream em blocos de 64 KiB, e; ser concebida de modo a exprimir o paralelismo potencial possível (entre as operações de I/O e o processamento de múltiplos blocos de *bytes*) de modo a que possa utilizar todos os processadores disponíveis.

5. [4] Usando a TPL (*Task Parallel Library*) e/ou os métodos assíncronos do .NET *Framework*, implemente o seguinte método estático da classe AsyncUtils.

```
public static Task<long> StreamFindByteAsync(Stream stream, Predicate<byte> filter);
```

Este método calcula quantos *bytes* de um *stream* satisfazem o predicado especificado através do argumento filter. O método deve: processar os *bytes* do *stream* em blocos de 64 KiB, e; ser concebido de modo a exprimir o paralelismo potencial possível de modo a que possa utilizar todos os processadores disponíveis.

Para ler assincronamente informação do *stream* use o método ReadAsync da classe Stream, cujo assinatura é a seguinte:

```
Task<int> ReadAsync(byte[] buffer, int offset, int count);
```

O método devolve retornar o mais cedo possível, devolvendo uma instância da classe Task<long> para representar a computação em curso.

Duração: 2 horas e 30 minutos
ISEL, 25 de Junho de 2014