Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Verão de 2012/2013

Teste Final (1ª Época)

---

1. [2,5] Considere a classe `SpinCountdownEvent`, não *thread-safe*, apresentada a seguir em C#:

```
public class SpinCountdownEvent {
  private int count;

  public SpinCountdownEvent(int partners) {
    if (partners > 0) count = partners;
  }

  public void Signal(){
    if (count > 0)
      count--;
    else
      throw InvalidOperationException();
 }

  public void Wait() {
    if (count == 0) return;
    SpinWait sw = new SpinWait();
    do {
      sw.SpinOnce();
    } while (count != 0);
  }
}
```

   a) [2] Diga quais as razões pelas quais a classe `SpinCountdownEvent` não é *thread-safe*. Sem recorrer à utilização de primitivas de sincronização bloqueantes, apresente as alterações necessárias (escolhendo Java ou C#) para tornar esta classe *thread-safe*.

   b) [0,5] Considera adequada esta implementação para este tipo de sincronizador? Justifique.

2. [4,5] Considere o sincronizador `MessageResequencer<T>` cujo objectivo é sequenciar a recepção das mensagens que lhe são confiadas. Cada mensagem, com o tipo `T`, enviada para o sincronizador tem o número de sequência, que é definido através de um argumento da operação `Send`. (Os números de sequência são inteiros positivos, com origem em 1.) Por exemplo, se a última mensagem entregue pelo sincronizador (a uma *thread* que invoque a operação `Receive`) tiver o número de sequência 42, poderão existir *threads* receptoras bloqueadas, mesmo que o sincronizador já tenha na sua posse as mensagens com os números de sequência 44, 45 e 49. Assim, que for enviada para o sincronizador a mensagem com o número de sequência 43, passam a existir condições para serem entregues as mensagens com os números de sequência 43, 44 e 45. As *threads* que invocam a operação `Receive` ficam bloqueadas até que obtenham uma mensagem ou até que a respectiva espera seja interrompida. O sincronizador deverá rejeitar a recepção de uma mensagem (i.e., assinalar excepção) quando o seu número de sequência for inferior ao da última mensagem já entregues ou igual ao número de sequência de uma mensagem ainda não entregue. Usando os monitores disponíveis no Java ou na CLI implemente o sincronizador `MessageResequencer<T>`.

3. [4,5] Usando monitores do Java ou da CLI, implemente o sincronizador `MessageBroadcaster<T>` que promove a difusão das mensagens que lhe são entregues através da operação `Send`. O interesse na recepção de mensagens é manifestado através da chamada à operação `Receive`. A operação `Send` entrega a mensagem especificada a todas as *threads* em espera, retornando `true`, ou descarta a mensagem se não houver *threads* em espera e retorna `false`. A operação `Receive` bloqueia a *thread* invocante até que: (1) seja difundida a próxima mensagem; (2) a espera da *thread* seja interrompida, ou; (3) seja atingido o limite de tempo de espera especificado. (Tenha em consideração que cada mensagem só é difundida às *threads* que se encontram em espera no momento em que a mensagem é divulgada.)

4. [8,5] Considere a classe `WebRequest_`, cuja implementação se baseia na classe `System.Net.WebRequest`, e que tem a seguinte interface pública:

```
public class WebRequest_ {
  public WebRequest_(string url);
  public long GetContentLength();
  public IAsyncResult BeginGetContentLength(AsyncCallback ucb, object ust);
  public long EndGetContentLength(IAsyncResult iar);
  public async Task<long> GetContentLengthAsync();
}
```

A classe `WebRequest_` define métodos que permitem obter o atributo *content-length* da resposta ao pedido HTTP/GET que é dada pela *Web*, quando se especifica um URL particular. Existem três interfaces distintas para obter a mesma informação: (a) interface síncrona, através do método `GetContentLength`; (b) interface assíncrona, ao estilo APM, através dos métodos `BeginGetContentLength` e `EndGetContentLength`, e; (c) interface assíncrona, através do método C# assíncrono `GetContentLengthAsync`.

Considere também a classe `AggregateWebRequest_` que visa implementar uma funcionalidade semelhante à da classe `WebRequest_`, mas sobre um grupo de URLs. Neste caso, considere que o valor do atributo *content-length* é o resultado da soma dos atributos *content-length* de todas as respostas aos pedidos HTTP/GET dadas pelos servidores que responderem pelos URLs do grupo. A interface pública desta classe é a seguinte:

```
class AggregateWebRequest_ {
  public AggregateWebRequest_(string[] urls);
  public IAsyncResult BeginGetContentLength(AsyncCallback ucb, object ust);
  public long EndGetContentLength(IAsyncResult iar);
  public Task<long> GetContentLengthTPL();
  public async Task<long> GetContentLengthAsync();
}
```

Usando a classe `WebRequest_` e ignorando os erros que podem ocorrer nos acessos à *Web*, para simplificar a implementação, responda às seguintes perguntas:

a) [2,5] Implemente os métodos `BeginGetContentLength` e `EndGetContentLength`, que permitam obter a soma dos valores dos atributos *content-length*, usando uma interface ao estilo APM. (Simplifique a implementação da interface `IAsyncResult`, nomeadamente usando *locking* explícito para obter a necessária sincronização.)

b) [2,5] Usando a funcionalidade da TPL, implemente o método `GetContentLengthTPL` de modo a que este devolva o resultado através de uma instância do tipo `Task<long>` (e.g., usando o *future pattern*).

c) [2,5] Implemente o método `GetContentLengthAsync`, usando os métodos assíncronos disponíveis na linguagem C#.

d) [1] Escreva um programa de teste da classe `AggregateWebRequest_` que utilize o método `GetContentLengthTPL`. Este programa deve mostrar uma mensagem na consola assim que as operações assíncronas HTTP/GET tiverem sido iniciadas e, mais tarde, quando todas as operações estiverem concluídas, mostrar na consola o respectivo *content-length*.

Duração: 2 horas e 30 minutos

Carlos Martins

ISEL, 4 de Julho de 2013