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

Teste Final (2ª Época)

---

1. [2] Considere a classe `UnsafeRefCountedHolder<T>`, não *thread-safe*, apresentada a seguir em C#:

```
public class UnsafeRefCountedHolder<T> where T : class {
  private T _value;
  private int _refs;

  public UnsafeRefCountedHolder(T v) {
    _value = v;
    _refs = 1;
  }

  public void AddRef() {
    if (_refs == 0)
      throw new InvalidOperationException();
    _refs++;
  }

  public void ReleaseRef () {
    if (_refs == 0)
      throw new InvalidOperationException();
    if (--_refs == 0) {
      IDisposable disposable = _value as IDisposable;
      _value = null;
      if (disposable != null)
        disposable.Dispose();
    }
  }

  public T Value {
    get {
      if (_refs == 0)
        throw new InvalidOperationException();
     return _value;
    }
  }
}
```

   Diga quais as razões pelas quais a classe `UnsafeRefCountedHolfer<T>` 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*.

2. [4] Usando os monitores disponíveis nas linguagens C# ou Java, implemente o sincronizador *transient signal*, cuja interface pública é a semântica de sincronização se descreve a seguir.

```
public class TransientSignal {
  public void Wait();
  public void SignalAll();
}
```

   A chamada ao método `Wait` bloqueia a *thread* invocante até que ocorra a próxima chamada ao método `SignalAll` ou que a espera da *thread* seja cancelada por interrupção. O método `SignalAll` debloqueia todas as *threads* bloqueadas no sincronizador, não tendo qualquer efeito se não existirem *threads* em espera (semântica idêntica à da *condition variables* dos monitores).

3. [4] Usando os monitores disponíveis nas linguagens C# ou Java, implemente o sincronizador *blackboard*, cuja interface pública e semântica de sincronização se descreve a seguir.

```
public class Blackboard<T> {
  public void Write(T message);
  public void Clear();
  public bool Read(int timeout, out T message)
}
```

O sincronizador *blackboard* destina-se a suportar a divulgação de mensagens entre *threads*. O *blackboard* tem dois estados possíveis: estar limpo ou ter afixada uma mensagem. O método `Write` afixa a mensagem especificada no *blackboard*, substituindo a mensagem actual, se existir. O método `Clear` deixa o *blackboard* limpo. O método `Read` é usado para ler uma mensagem do *blackboard*, e bloqueia a *thread* invocante até obter a mensagem, expirar o limite de tempo de espera especificado ou a sua espera cancelada por interrupção. (As mensagens que afixadas no *blackboard* têm que ser obrigatoriamente entregues a todas as *threads* que se encontrem bloqueadas no momento da afixação, mesmo quando o método `Clear` é invocado imediatamente a seguir ao método `Write` que afixou a mensagem.)

4. [6] Considere a operação "*Count If*" que aplicada a um *byte stream* devolve o número de *bytes* do *stream* que satisfazem ao predicado especificado. Pretende-se implementar a operação "*Count If*" com os dois tipos de interface assíncrona, disponíveis no .NET *Framework* (i.e., ao estilo APM e com base num método assíncrono). Ambas as implementações devem paralelizar o processamento do último bloco de *bytes* lidos do *stream* com a leitura do próximo bloco de *bytes*. Use os métodos da classe `System.IO.Stream`, mostrados a seguir, para realizar leituras assíncronas e, para simplificar a implementação, ignore a ocorrência de excepções:

```
IAsyncResult BeginRead(byte[] buffer, int offset, int count, AsyncCallback ucb,
                       object ustate);
int EndRead(IAsyncResult iar);
Task<int> ReadAsync(byte[] buffer, int offset, int count);
```

   a) [3] Implemente os seguintes métodos, que implementam a operação *"Count If"* ao estilo APM:

```
static IAsyncResult BeginCountIf(Stream source, Predicate<byte> filter,
                                 AsyncCallback, ucb, object ustate);
static long EndCountIf(IasyncResult iar);
```

   b) [2] Implemente a operação *"Count If"* com um método assíncrono, com a seguinte assinatura:

```
static async Task<long> CountIfAsync(Stream source, Predicate<byte> filter);
```

   c) [1] Discuta sucintamente quais são as vantagens e os inconveniente de cada uma das interfaces assíncronas implementadas anteriormente. Compare as duas soluções relativamente ao desempenho.

5. [1] Considere o seguinte método, escrito em C#:

```
static int Compute(Func<int> a, Func<int> b, Func<int> c, Func<int> d) {
  return (a() + b() - c()) * d();
}
```

   Usando os mecanismos disponíveis na TPL, escreva a função `ParallelCompute`, com a mesma assinatura e semântica do método `Compute`, que explore o paralelismo possível e que optimize o número de *threads* mobilizadas.

6. [3] Usando os mecanismos disponíveis na TPL, implemente o método `SpeculativeInvoke`, com a assinatura e especificação que se descreve a seguir.

```
static Task<TResult> SpeculativeInvoke(
                                 params Func<CancellationToken, TResult>[] functions);
```

   Este método promove a execução em paralelo de todas as funções passadas através do parâmetro `functions`. O método retorna uma instância de `Task<TResult>` cujo valor subjacente será o `TResult` calculado pela função que primeiro concluir a execução. Após ter sido obtido o resultado, a execução das funções ainda em execução deverá ser cancelada e de deverão ser observadas todas as excepções que subjacentes ao cancelamento.

Duração: 2 horas e 30 minutos

Carlos Martins

ISEL, 19 de Julho de 2013