Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente
Teste Global da Época de Recurso, Inverno de 2018/2019

---

1. [2,5] Considere a definição em C# da classe **`UnsafeCountdownEvent`**, como uma tentativa para implementar o sincronizador *count down event* implementado pelo .NET *framework*.

```
public class UnsafeCountdownEvent {
  private ManualResetEventSlim waitEvent = new ManualResetEventSlim(false);
  private int count;

  public UnsafeCountdownEvent(int start) {
    if (start < 0) throw new ArgumentException("start");
    count = start;
  }

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

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

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

   Esta classe ilustra um algoritmo que implementa o sincronizador *count down event*, contudo não é *thread safe*. Sem usar *locks*, implemente, em C# ou *Java* , uma versão *thread safe* desta classe (**`SafeCountDownEvent`**).

2. [4] Implemente em C# ou *Java*, com base nos monitores implícitos ou explícitos, o sincronizador *packet builder* que agrega mensagens entregues individualmente por *threads* produtoras em pacotes que disponibiliza às *threads* consumidoras, e cuja interface pública em C# é a seguinte:

```
public class PacketBuilder<T> {
  public PacketBuilder(int packetSize);
  public void PutMsg(T message);
  public List<T> TakeMsgPacket(int timeout);
}
```

   O método **`PutMsg`** é invocado pelas *threads* produtoras para entregar mensagens ao sincronizador e nunca bloqueia a *thread* invocante. Quando a mensagem enviada com este método completar um pacote (com **`packetSize`** mensagens), e houver alguma *thread* bloqueada, o pacote de mensagens deve ser entregue à *thread* que se encontra bloqueada há mais tempo. As mensagens enviadas através do método **`PutMsg`** nunca devem ser descartadas. O método **`TakeMsgPacket`** permite obter o próximo pacote de mensagens, e termina: (a) com sucesso, devolvendo a referência para a lista que contém as mensagens do pacote; (b) retornando **`null`**, se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`ThreadInterruptedException`** quando a espera da *thread* é Interrompida.

3. [4] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *transfer queue*, para suportar a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico **`E`**. A interface pública deste sincronizador, em C#, é a seguinte:

```
public class TransferQueue<E> where E : class {
  public void Put(E message);
  public bool Transfer(E message, int timeout);
  public E Take(int timeout);
}
```

O método **`Put`** entrega uma mensagem à fila e nunca bloqueia a *thread* invocante. O método **`Transfer`** entrega uma mensagem à fila, com garantia da respectiva recepção, pelo que pode bloquear a *thread* invocante até que a mensagem seja recebida por uma *thread* consumidora, e termina: (a) com sucesso, retornando **`true`** quando a mensagem é recebida; (b) retornando **`false`** se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`ThreadInterruptedException`** quando a espera da *thread* é Interrompida. Quando o método **`Transfer`** termina sem sucesso, a respectiva mensagem deve ser descartada. O método **`Take`** recebe a próxima mensagem da fila, e termina: (a) com sucesso, retornando a referência para a mensagem recebida; (b) retornando **`null`** se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`ThreadInterruptedException`** quando a espera da *thread* é Interrompida. A implementação do sincronizador deve-se optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

4. [6] Considere o seguinte método:

```
int[] Oper(int[] a, int[] b) {
  if(a.Length != b.Length) throw new ArgumentException();
  var res = new int[a.Length];
  for (int i = 0; i < a.Length; ++i) {
    res[i] = Join(Map(a[i]), Map(b[i]));
  }
  return res;
}
```

   a. [2] Comente a afirmação: uma instância do tipo **`Task`** definido pelo .NET *framework* está sempre associada a uma operação em realização numa *worker thread* do *default task scheduler* (*thread pool*).

   b. [4] Os métodos **`Map`** e **`Join`** são funções sem efeitos colaterais e passível de múltiplas execuções em paralelo. Realize uma versão assíncrona do método **`Oper`** seguindo o padrão TAP (*Task-based Asynchronous Pattern*) usando a TPL e/ou os métodos assíncronos do C#. Assuma que tem disponível versões TAP dos métodos **`Map`** e **`Join`**. Tire partido do paralelismo potencial existente, valorizando-se soluções que iniciem as operações **`Map`** e **`Join`** o mais cedo possível.

5. [3,5] Considere o método **`VerifiedMean que`** calcula a média dos pesos associados a uma colecção de itens do tipo **`T`**. O peso de cada item é obtido com a função **`evaluator`** e verificado com a função **`ok`**, que retorna **`false`** no caso do peso passado como argumento não ser considerado aceitável. Se o total de pesos rejeitados pela função **`ok`** ultrapassar o limite especificado com o argumento **`maxBads`**, o método **`VerifiedMean`** deve lançar **`InvalidOperationException`**. Assumindo que é na execução das funções **`evaluator`** e **`ok`** que se concentra a maior parte do processamento e usando a *Task Parallel Library*, implemente uma versão do método **`VerifiedMean`** que tire partido dos múltiplos *cores* de processamento disponíveis (**`ParallelVerifiedMean`**).

```
public static double VerifiedMean<T>(IEnumerable<T> col,
                                      Func<T, double> evaluator,
                                      Func<double, bool> ok,
                                      int maxBads)
```

Duração: 2 horas e 30 minutos
ISEL, 23 de janeiro de 2019