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

---

1. [3] Considere a definição em *Java* da classe **`UnsafeSpinExpirableLazy`**, como uma tentativa para implementar o sincronizador *expirable lazy*, cuja implementação baseada em monitores foi solicitada na primeira série de exercícios.

```
public class UnsafeSpinExpirableLazy {
  private class StateHolder {
    T value;
    long expiresAt;
    StateHolder(T v, long timeToLive) {
      value = v; expiresAt = System.currentTimeMillis() + timeToLive;
    }
  }
  private final StateHolder COMPUTING = new StateHolder(null, 0L);
  private final long timeToLive;
  private final Supplier<T> supplier;
  private StateHolder state;
  UnsafeSpinExpirableLazy(Supplier<T> supplier, long timeToLive) {
    this.supplier = supplier; this.timeToLive = timeToLive;
  }
  public T getValue() {
    do {
      if (state != null && state != COMPUTING) {
        if (state.expiresAt > System.currentTimeMillis()) return state.value;
        else state = null;  // value has expired, reset state and continue
      }
      if (state == COMPUTING) {   // if the value is being calculated, spin and retry get
        Thread.yield(); continue;
      } else if (state == null) { // if no one is calculating, try to calculate it
        state = COMPUTING;
        T newValue = supplier.get(); // compute the new value
        state = new StateHolder(newValue, timeToLive);  // publish the new value
        return newValue;
      }
    } while (true);
  }
}
```

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

2. [4] Implemente, em *Java* ou C#, como base nos monitores implícitos ou explícitos, o sincronizador *broadcast* que promove a difusão de mensagens entre *threads* e cuja interface pública em C# é a seguinte:

```
public class Broadcast<M> where M : class {
  public bool Send(M message);
  public M Receive(int timeout);
}
```

   O interesse das *threads* na recepção das mensagens difundidas é manifestado através da invocação da operação **`Receive`**. A operação **`Send`** entrega a mensagem especificada com o argumento **`message`** a todas as *threads* que se encontram em espera, retornando **`true`**, ou descarta a mensagem caso não existam *threads* em espera, retornando **`false`**. A operação **`Receive`** bloqueia a *thread* invocante aguradando a difusão da próxima mensagem, e termina: (a) com sucesso, retornando a mensagem recebida; (b) devolvendo **`null`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`ThreadInterruptedException`**, se o

bloqueio da *thread* for interrompido. (Tenha em consideração que as mensagens apenas devem ser entregues às *threads* que se encontram bloqueadas pela operação **`Receive`** no momento da respectiva difusão.)

3. [4] Implemente, em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *keyed event*, implementado pelo sistema operativo *Windows*, cuja interface pública em C# é a seguinte:

```
public class KeyedEvent {
  public bool Release(object key, int timeout);
  public bool Wait(object key, int timeout);
}
```

   As operações **`Release`** e **`Wait`** são ambas bloqueantes; a operação **`Release`** bloqueia a *thread* invocante até que seja invocada, por outra *thread*, a operação **`Wait`** especificando a mesma chave; o simétrico acontece com a operação **`Wait`**. As operações **`Release`** e **`Wait`** podem bloquear a *thread* invocante, e terminam: (a) devolvendo **`true`** se ocorrer a chamada à operação complementar; (b) devolvendo **`false`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido. As *threads* devem ser desbloqueadas com disciplina *first-in-first-out* (FIFO), isto é, a operação **`Release`** deve desbloquear a *thread* que se encontra bloqueada há mais tempo por **`Wait`** e a operação **`Wait`** deverá desbloquear a *thread* que se encontra bloqueada há mais tempo por **`Release`**.

4. [5,5] A interface **`IServices`** define um conjunto de operações síncronas que envolvem comunicação com sistemas externos com latências da ordem dos segundos, que usam os tipos **`I`**, **`O`** e **`U`**, previamente definidos. A operação **`Oper2`** não altera o estado (local ou dos sistemas externos); a execução das operações **`Oper1`** e **`Oper2`** pode resultar em excepção. O método **`Run`** executa uma sequência destas operações, produzindo o resultado final ou excepção.

```
public class Execute {
  public interface IServices {
    U Oper1(I i);
    O Oper2(U u, int i);
    void Oper3(U u);
  }
  public static O[] Run(IServices s, I ui, int n) {
    U u = s.Oper1(ui);
    O[] result = new O[n];
    try { for (int i = 0; i < n; i++) result[i] = s.Oper2(u, i); } finally { s.Oper3(u); }
    return result;
  }
}
```

   a. [1,5] Diga sucintamente quais são as principais vantagens da programação assíncrona e indique quais as características das aplicações em que essas vantagens são mais evidentes.

   b. [4] A classe **`TAPExecute`** será a variante assíncrona de **`Execute`**, ao estilo *Task-based Asynchronous Pattern* (TAP). Usando a funcionalidade oferecida pela *Task Parallel Library* (TPL) ou pelos métodos assíncronos do C#, implemente o método **`RunAsync`**, que usa a interface **`ITAPServices`** (variante TAP de **`IServices`** que não tem de apresentar). A implementação deve exprimir o paralelismo potencial que existe no método **`Run`** e tratar as excepções possíveis.

5. [3,5] Usando os mecanismos disponíveis na *Task Parallel Library*, implemente uma versão do método estático **`MapSelected`** que tire partido de todos os *cores* de processamento disponíveis no computador. Este método retorna o resultado da transformação (*mapping*) de todos os elementos de um enumerado que satisfaçam um dado predicado. Tenha em consideração que o processamento dos elementos do enumerado pode ser executado em paralelo, que a operação de transformação é aquela que concentra a maior componente de processamento, que a operação do método deve poder ser cancelada e que a ordem dos elementos na lista resultante não é relevante.

```
public static List<O> MapSelected<I,O>(IEnumerable<I> items, Predicate<I> selector,
                                       Func<I,O> mapper, CancellationToken ctoken);
```

Duração: 2 horas e 30 minutos
ISEL, 25 de Junho de 2019