Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global da Época Normal, Inverno de 2019/2020

---

1. [2] Considere a definição em *Java* da classe **`UnsafeSpinExchanger`,** como uma tentativa para implementar um sincronizador com a semântica idêntica ao *exchanger* disponível no *Java*.

```
class UnsafeSpinExchanger<T> {
  private class Holder {
    T first, second;
    boolean done;
    Holder(T f) { first = f; }
  }
  private Holder xchg = null;        // the exchange spot
  public T exchange(T myData, long timeout) {
    if (xchg == null) {
      var h = new Holder(myData);
      xchg = h;
      TimeoutHolder th = new TimeoutHolder(timeout);
      while (!h.done && th.value() > 0) Thread.yield();
      if (h.done) return h.second;
      if (xchg == h) {
        xchg = null; return null;
      }
      while (!h.done) Thread.yield();
      return h.second;
    } else {
      var h = xchg;
      xchg = null;
      h.second = myData;
      h.done = true;
      return h.first;
    }
  }
}
```

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

2. [5] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, uma variante do sincronizador *message queue*. A interface pública, na linguagem *Java*, é a seguinte:

```
public class MessageQueue<E> {
  public void put(E message);
  public List<E> get(int nOfMessages, int timeout) throws InterruptedException;
}
```

   O método **`put`** entrega uma mensagem à fila e nunca bloqueia a *thread* invocante. O método **`get`** permite receber **`nOfMessages`** mensagens da fila, e termina: (a) com sucesso, retornado a referência para uma lista com as **`nOfMessages`** mensagens recebidas; (b) retornando **`null`** se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`InterruptedException`** quando a espera da *thread* é interrompida. A receção das mensagens deve seguir a política FIFO (*first in first out*): a primeira *thread* a realizar **`get`** deve ser a primeira a ter o seu pedido satisfeito. Um chamada a **`get`** que retorne sem sucesso (*timeout* ou excepção) não consome elementos da fila.

3. [5] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *bounded bucket*, que viabiliza a entrega em lote (i.e., *bucket*) de itens de dados inseridos individualmente. A interface pública deste sincronizador, na linguagem *Java*, é a seguinte:

```
public class BoundeBucket<T> {
  public BoundedBucket(int capacity);
  public boolean put(T ite, long timeout) throws InterruptedException;
  public List<T> takeAll(long timeout) throws InterruptedException;
}
```

A capacidade do *bucket* é especificada na construção com o argumento **`capacity`**. O método **`put`** entrega um item de dados ao *bucket*, bloqueando a *thread* invocante quando tiver sido excedida a respectiva capacidade, e retornando: a) o valor **`true`**, se o item de dados foi entregue ao *bucket*; b) o valor **`false`**, se for excedido o limite especificado para o tempo de espera, e; c) lançando **`InterruptedException`** quando a espera da *thread* é Interrompida. O método **`takeAll`** permite recolher um lote de itens de dados, bloqueando a *thread* invocante até que exista pelo menos um item disponível, retornando: a) a referência para lista com as referência para os itens de dados; b) o valor **`null`**, se for excedido o limite especificado para o tempo de espera, e; c) lançando **`InterruptedException`** quando a espera da *thread* é Interrompida.

4. [5] Considere o seguinte método:

```
public R[] Compute(T[] elems)  {
  var res = R[elems.Length];
  for (int i = 0 ; i < elems.Length ; i++)
    res[i] = Oper(elems[i]);
  return res;
}
```

O método **`Oper`** é uma função sem efeitos colaterais e passível de múltiplas execuções em paralelo. Esporadicamente pode lançar uma excepção (e.g. devido a erro de comunicação). Realize uma versão assíncrona do método **`Compute`** seguindo o padrão TAP (*Task-based Asynchronous Pattern*) usando a TPL e/ou os métodos assíncronos do C#. Assuma que tem disponível uma versão TAP do método **`Oper`**. Tire partido do paralelismo potencial existente. A versão assíncrona deve tolerar erros na operação **`Oper`**, tentando realizar de novo a operação para o mesmo parâmetro. Contudo, não devem ser realizadas mais do que **`maxRetries`** retentativas para cada elemento de **`elems`**, onde **`maxRetries`** é um parâmetro de entrada. Caso seja excedido o número de tentativas, devem ser canceladas todas as operações pendentes.

5. [3] No método **`MapReduce`**, apresentado a seguir, as invocações a **`Map`** podem decorrer em paralelo, o que seria vantajoso já que é nessa operação que se concentra a maior componente de processamento. O método **`Reduce`** implementa a operação, comutativa e associativa, que agrega os resultados, sendo **`null`** o respectivo elemento neutro. A operação de redução pode transbordar, retornando **`Result.OVERFLOW`**, situação em que o método **`MapReduce`** deverá retornar rapidamente lançando a excepção **`OverflowException`**. Tirando partido da *Task Parallel Library*, implemente o método **`ParallelMapReduce`** que faça invocações paralelas ao método **`Map`** de modo a tirar partido de todos os *cores* de processamento disponíveis.

Nota: Tenha em consideração que no *overload* do método **`Parallel.ForEach`** não é possível desencadear a paragem do *parallel loop* a partir do *lambda* que agrega os resultados parciais com o resultado global.

```
static Result MapReduce(IEnumerable<Data> indata, CancellationToken ctoken)  {
  Result result = null;
  foreach (var datum in indata) {
    ctoken.ThrowIfCancellationRequested();
    if ((result = Reduce(result, Map(datum)) == Result.OVERFLOW)
      throw new OverflowException();
  }
  return result;
}
```

Duração: 2 horas e 30 minutos
ISEL, 14 de janeiro de 2020