Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 1ª Época, Verão de 2015/2016

---

1. [2.5] Considere a classe `UnsafeCountDownLatch`, cuja implementação, em *Java* é apresentada a seguir:

```
class UnsafeCountDownLatch {
  private static final int WAIT_GRAIN = 10;
  private int count;

  public UnsafeCountDownLatch(int initial) { if (initial > 0) count = initial; }

  public void signal() throws IllegalStateException {
    if (count > 0) count--; else throw new IllegalStateException();
  }

  public boolean await(long timeout) throws InterruptedException {
    if (timeout < 0) timeout = Long.MAX_VALUE;
    for (; count > 0 && timeout > 0; timeout -= WAIT_GRAIN)
      Thread.sleep(WAIT_GRAIN);
    return count == 0;
  }
}
```

   Esta implementação reflete a semântica de sincronização do *count down latch*, contudo não é *thread-safe*. Usando técnicas de sincronização *non-blocking*, implemente em *Java* uma versão *thread-safe* deste sincronizador.

2. [3,5] Usando os monitores disponíveis nas linguagens C# ou *Java*, implemente o sincronizador *transient signal*, cuja interface pública, em *Java*, e a semântica de sincronização se descrevem a seguir.

```
class TransientSignal {
  public boolean await(long timeout) throws InterruptedException;
  public void signal();
  public void signalAll();
}
```

   A chamada ao método **`await`** bloqueia sempre a *thread* invocante até que: (a) ocorra uma notificação por invocação do método **`signal`** ou do método **`signalAll`**; (b) expire o limite de tempo de espera especificado, ou; (c) o bloqueio da *thread* seja interrompido. O método **`signal`** desbloqueia a *thread* que se encontrar bloqueada há mais tempo (FIFO). O método **`signalAll`** desbloqueia todas as *threads* que se encontrem bloqueadas pelo método **`await`**. As chamadas aos métodos **`signal`** e **`signalAll`** não têm qualquer efeito quando não existirem *threads* bloqueadas (semântica idêntica à das *condition variables* dos monitores).

3. [4] Usando os monitores disponíveis na linguagem *Java*, Implemente o sincronizador *message sequencer*, cuja interface pública e a semântica de sicronizaçao se descrevem a seguir:

```
class MessageSequencer<T> {
  public void send(int sn, T msg) throws IllegalArgumentException;
  public T receive(long timeout) throws InterruptedException;
}
```

   O objectivo deste sincronzador é forçar a sequenciação na recepção das mensagens que lhe são confiadas. Cada mensagem do tipo **`T`**, enviada para o sincronizador, tem associado um número de sequência (inteiro positivo, com origem em 1) que é definido pelo argumento **`sn`** do método **`send`**. Por exemplo, se a última mensagem entregue pelo sincronizador (a uma *thread* que tenha invocado a operação **`receive`**) tinha 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. Nesta situação, quando 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: (a) obtenham uma mensagem; (b) expire o limite de tempo de espera especificado, ou; (c)

o bloqueio da *thread* seja interrompido.
O sincronizador deverá rejeitar as mensagens (i.e., lançar **`IllegalArgumentException`**) cujo número de sequência for inferior ao da última mensagem já entregue ou igual ao número de sequência de uma mensagem ainda por entregar.
NOTA: Utilize o tipo **`java.util.PriorityQueue<T>`**. Este tipo implementa uma fila com a garantia de que o elemento que se encontra à cabeça é sempre o de menor ordem de entre os presentes na fila, de acordo com o determinado pela implementação da interface **`Comparator<T>`** que é passada para o construtor de **`PriorityQueue<T>`**.

4. [7] A interface **`Services`** define os serviços síncronos disponibilizados por uma organização que oferece a execução de diversos serviços em servidores localizados em diferentes áreas geográficas (SaaS). O método **`PingServer`** responde a um pedido de *ping*, devolvendo o **`Uri`** do respectivo servidor. O método **`ExecService`** executa, no servidor especificado através do parâmetro **`server`**, o serviço especificado pelos tipos genéricos **`S`** (serviço) e **`R`** (resposta)**.** O método **`ExecOnNearServer`** usa as operações de **`Services`** para executar de forma síncrona o serviço especificado, no servidor que primeiro responder ao serviço **`PingServer`**, isto é, aquele que se considera ser o servidor mais próximo.

```
public class Exec {
  public interface Services<S, R> {
    Uri PingServer(Uri server);
    R ExecService(Uri server, S service);
  }
  public R ExecOnNearServer<S, R>(Services<S, R> svc, Uri[] servers, S service);
}
```

   a. [3] A classe **`APMExec`** será a variante assíncrona de **`Exec`** ao estilo *Asynchronous Programming Model* (APM). Implemente os métodos **`BeginExecOnNearServer`** e **`EndExecOnNearServer`** que usam a interface **`APMServices`** (variante APM de **`Services`** que não tem de apresentar).
   NOTA: não pode usar a TPL e só se admitem esperas de controlo dentro das operações **`End`**, estritamente onde o APM o exige.

   b. [4] A classe **`TAPExec`** será a variante assíncrona de **`Exec`**, ao estilo *Task based Asynchronous Pattern* (TAP). Usando a funcionalidade oferecida pela *Task Parallel Library* (TPL) ou pelos métodos **`async`** do C#, implemente o método `ExecOnNearServerAsync`, que usa a interface **`TAPServices`** (variante TAP de **`Services`** que não tem de apresentar).
   NOTA: na implementação não se admite a utilização de operações com bloqueios de controlo.

5. [3] No método **`MapAggregate`**, apresentado a seguir, as invocações a **`Map`** podem decorrer em paralelo, o que seria vantajoso já que é nessa operação se concentra a maior componente de processamento. O método **`Aggregate`** implementa a operação, comutativa e associativa, que agrega os resultados, sendo o respectivo elemento neutro produzindo pela expressão **`new Result()`**. A operação de agregação pode retornar *overflow*, situação em que o método **`MapAggregate`** deverá retornar rapidamente essa indicação. Tirando partido da *Task Parallel Library*, apresente uma versão do método **`MapAggregate`** que faça invocações paralelas ao método **`Map`** de modo a tirar partido de todos os *cores* de processamento disponíveis.
   NOTA: considere que o método **`Aggregate`** retorna **`Result.OVERFLOW`** quando um dos seus argumentos já for *overflow*.

```
public static Result MapAggregate(IEnumerable<Data> data) {
  Result result = new Result();
  foreach (var datum in data) {
    result = Aggregate(result, Map(datum));
    if (result.Equals(Result.OVERFLOW)) break;
  }
  return result;
}
```

Duração: 2 horas e 30 minutos
ISEL, 21 de Junho de 2016