# Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Inverno de 2017/2018

Série de Exercícios 1

Escreva classes *thread-safe* para implementar os sincronizadores especificados utilizando os monitores implícitos da plataforma .NET e/ou os monitores implícitos ou os monitores explícitos disponíveis no *Java*. Para cada sincronizador, apresente pelo menos um dos programas ou testes que utilizou para verificar a correção da respectiva implementação.

1. Implemente em C# a classe **`ExpirableLazy<T>`** com a seguinte interface pública

```
public class ExpirableLazy<T> where T:class {
  public ExpirableLazy(Func<T> provider, TimeSpan timeToLive);
  public T Value {get;} // throws InvalidOperationException, ThreadInterruptedException
}
```

   Esta classe implementa uma versão da classe **`System.Lazy<T>`**, pertencente à plataforma .NET, *thread-safe* e com limitação no tempo de vida, especificado através do parâmetro **`timeToLive`**, do valor calculado.
   O acesso à propriedade **`Value`** deve ter o seguinte comportamento: (a) caso o valor já tenha sido calculado e o seu tempo de vida ainda não tenha expirado, retorna esse valor; (b) caso o valor ainda não tenha sido calculado ou o tempo de vida já tenha sido ultrapassado, inicia o cálculo chamando **`provider`** na própria *thread* invocante e retorna o valor resultante; (c) caso já existe outra *thread* a realizar esse cálculo, espera até que o valor esteja calculado; (d) lança **`ThreadInterruptedException`** se a espera da *thread* for interrompida. Caso a chamada a **`provider`** resulte numa excepção: (a) a chamada a **`Value`** nessa *thread* deve resultar no lançamento dessa excepção; (b) se existirem outras *threads* à espera do valor, deve ser seleccionada uma delas para a retentativa do cálculo através da função **`provider`**. Não existe limite no número de retentativas. O tempo de vida inicia-se quando o valor é retornado da função **`provider`**.

2. Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *transfer queue* que suporta a comunicação entre *threads* produtoras e consumidoras e cuja interface pública em C# é a seguinte:

```
public class TransferQueue<T> {
  public void Put(T msg);
  public bool Transfer(T msg, int timeout);  // throws ThreadInterruptedException
  public bool Take(int timeout, out T rmsg); // throws ThreadInterruptedException
}
```

   O método **`Put`** entrega uma mensagem à fila, e nunca bloqueia a *thread* invocante. O método **`Transfer`** entrega uma mensagem à fila e aguarda que a mesma seja recebida por outra *thread* (via método **`Take`**) e termina: (a) devolvendo **`true`**, se a mensagem for recebida por outra *thread*; (b) devolvendo **`false`**, se expirar o limite do tempo de espera especificado sem que a mensagem seja recebida por outra *thread*, ou; (c) lançando **`ThreadInterruptedException`** quando a espera da *thread* for interrompida. (Quando o método **`Transfer`** retorna sem sucesso, a respectiva mensagem não deve ficar na fila.) O método **`Take`** permite receber uma mensagem da fila e termina: (a) devolvendo **`true`** se for recebida uma mensagem, sendo esta devolvida através do parâmetro **`rmsg`**; (b) devolvendo **`false`**, se expirar o limite do tempo de espera especificado, ou; (c) lançando **`ThreadInterruptedException`** quando a espera da *thread* é interrompida. Na implementação do sincronizador deve procurar minimizar o número de comutações de *threads* em todas as circunstâncias, usando as técnicas estudadas na unidade curricular.

3. Implemente em C# o sincronizador **`Pairing<T,U>`** com a seguinte interface pública:

```
public class Pairing<T, U> {
  // throws ThreadInterruptedException, TimeoutException
  public Tuple<T,U> Provide(T value, int timeout);
  public Tuple<T,U> Provide(U value, int timeout);
}
```

   Os métodos **`Provide`** entregam um valor ao sincronizador, ficando a *thread* invocante bloqueada à espera de um emparelhamento, que ocorre quando estiver presente um valor do tipo **`T`** e um valor do tipo **`U`**. Nesse estado, as duas *threads* responsáveis pela entrega retomam a sua execução, retornado ambas o mesmo tuplo contendo ambos os valores. Cada valor entregue ao sincronizador não poderá ser usado em mais do que um emparelhamento. Se a chamada do método **`Provide`** terminar com o lançamento de excepção (*timeout* ou interrupção), então o valor entregue não pode ser usado em nenhum emparelhamento.

4. Implemente em *Java*, com base nos monitores implícitos ou explícitos, o sincronizador *simple thread pool executor*, que executa os comandos que lhe são submetidos numa das *worker threads* que o sincronizador cria e gere para o efeito. A interface pública deste sincronizador é a seguinte:

```
public class SimpleThreadPoolExecutor {
  public SimpleThreadPoolExecutor(int maxPoolSize, int keepAliveTime);
  public boolean execute(Runnable command, int timeout) throws InterruptedException;
  public void shutdown();
  public boolean awaitTermination(int timeout) throws InterruptedException;
}
```

O número máximo de *worker threads* (**`maxPoolSize`**) e o tempo máximo que uma *worker thread* pode estar inactiva antes de terminar (**`keepAliveTime`**) são passados com argumentos para o construtor da classe **`SimpleThreadPoolExecutor`**. A gestão, por parte do sincronizador, das *worker threads* deve obedecer aos seguintes critérios: (1) se o número total de *worker threads* for inferior ao limite máximo especificado, é criada uma nova *worker thread* sempre que for submetido um comando para execução e não existir nenhuma *worker thread* disponível; (2) as *worker threads* deverão terminar após decorrerem mais do que **`keepAliveTime`** milésimos de segundo sem que sejam mobilizadas para executar um comando; (3) o número de *worker threads* existentes no *pool* em cada momento depende da actividade deste e pode variar entre zero e **`maxPoolSize`**.

As *threads* que pretendem executar funções através do *thread pool executor* invocam o método **`execute`**, especificando o comando a executar com o argumento **`command`**. Este método pode bloquear a *thread* invocante, pois tem que garantir que o comando especificado foi entregue a uma *worker threads* para execução, e pode terminar: (a) normalmente, devolvendo **`true`**, se o comando foi entregue para execução; (b) excepcionalmente, lançando a excepção **`RejectedExecutionException`**, se o *thread pool* se encontrar em modo *shutting down*; (c) excepcionalmente, devolvendo **`false`**, se expirar o limite de tempo especificado com **`timeout`** sem que o comando seja entregue a uma *worker thread*, ou; (d) excepcionalmente, lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido.

A chamada ao método **`shutdown`** coloca o executor em modo *shutting down* e retorna de imediato. Neste modo, todas as chamadas ao método **`execute`** deverão lançar a excepção **`RejectedExecutionException`**. Contudo, todas as submissões para execução feitas antes da chamada ao método **`shutdown`** devem ser processadas normalmente.

O método **`awaitTermination`** permite à *thread* invocante sincronizar-se com a conclusão do processo de *shutdown* do executor, isto é, até que sejam executados todos os comandos aceites e que todas as *worker threads* activas terminem, e pode terminar: (a) normalmente, devolvendo **`true`**, quando o *shutdown* do executor estiver concluído; (b) excepcionalmente, devolvendo **`false`**, se expirar o limite de tempo especificado com o argumento **`timeout`**, sem que o *shutdown* termine, ou; (c) excepcionalmente, lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido.

A implementação do sincronizador deve optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

Data limite de entrega: 31 de Outubro de 2017

ISEL, 2 de Outubro de 2017