Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Verão de 2016/2017

Série de Exercícios 1

Escreva classes *thread-safe* para realizar os sincronizadores especificados. Para cada sincronizador, apresente pelo menos um dos programas ou testes que utilizou para verificar a correção da respectiva implementação.

1. Implemente em C# ou *Java*, com base nos monitores implícitos ou explícitos (*Java*), a classe `ThottledRegion` com a seguinte interface pública:

   ```
   public class ThrottledRegion {
     public ThrottledRegion (int maxInside, int maxWaiting, int waitTimeout);
     public bool TryEnter(int key);     // throws ThreadInterruptedException
     public void Leave(int key);
   }
   ```

   O objectivo desta classe é controlar o número de *threads* que estão simultaneamente dentro de uma região de código delimitada pelas chamadas aos métodos `TryEnter` e `Leave`, para cada valor particular de chave. Por exemplo, pode ser usada para limitar o número de pedidos simultâneos que um servidor HTTP pode estar a executar para o mesmo utilizador (usando o identificador de utilizador como chave). Além de controlar o número de *threads* dentro da região, limita também o número de *threads* que podem estar em espera. Mais uma vez, no cenário de um servidor HTTP é por vezes preferível recusar imediatamente um pedido a colocá-lo em espera (com o consequente consumo de recursos como memória ou ligações TCP).

   Em qualquer momento do tempo e para qualquer chave: (a) não podem estar mais do que `maxInside` *threads* dentro da zona protegida pela mesma chave; (b) não podem estar mais do que `maxWaiting` *threads* à espera de entrar na zona protegida pela mesma chave; (c) uma *thread* não poderá esperar mais do que `waitTimeout` milésimos de segundo para entrar na zona protegida. A política de entrada nas zonas protegidas é FIFO (*first-in-first-out*). O método `TryEnter` retorna se a entrada foi feita com sucesso. Note-se que a entrada pode não ter sucesso devido à ocorrência de *timeout*, a ter sido excedido o número máximo de *threads* em espera ou ainda por ter sido interrompido o bloqueio da *thread*. Nas primeiras duas situações o método `TryEnter` retorna `false` e na última termina com o lançamento de `ThreadInterruptedException`.

2. Implemente em C# ou *Java*, com base nos monitores implícitos ou explícitos (*Java*), a classe `Exchanger<T>`, que suporta a troca de mensagens, definidas por instâncias do tipo genérico `T`, entre pares de *threads*, com a seguinte interface pública:

   ```
   public class Exchanger<T> where T : class {
     public T Exchange(T mine, int timeout);  // throws ThreadInterruptedException
   }
   ```

   O uníco método, `Exchange`, é invocado pelas *threads* para oferecer uma mensagem (`mine`) e receber, através do valor de retorno do método, a mensagem oferecida pela *thread* com que emparelharam. Quando a troca de mensagens não pode ser realizada de imediato (porque não existe ainda outra *thread* bloqueada), a *thread* que invova o método `Exchange` fica bloqueada até que: (a) outra *thread* invoque o método `Exchange`, devolvendo o método a mensagem oferecida pela outra *thread*; (b) expire o limite do tempo de espera especificado, situação em que o método devolve `null`, ou; (c) a espera seja interrompida, terminado o método com o lançamento de `ThreadInterruptedException`.

3. Implemente em C# ou *Java*, com base nos monitores implícitos ou explícitos (*Java*), a classe `RetryLazy<T>`, com a seguinte interface pública:

   ```
   public class RetryLazy<T> where T:class {
     public RetryLazy(Func<T> provider, int maxRetries);
     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 tolerância a falhas esporádicas (e.g., a inexistência momentânea de ligações num *pool* de ligações a bases de dados). Esta tolerância é conseguida através da retentativa de cálculo do valor em diferentes *threads*. O acesso à propriedade `Value` deve ter o seguinte comportamento: (a) caso o valor já tenha sido calculado, retorna esse valor; (b) caso o valor ainda não tenha sido calculado, e o número máximo de tentativas (especificado com `maxRetries`) não tenha sido excedido, inicia esse 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 ou o número de tentativas seja excedido; (d) lança `ThreadInterruptedException` se a espera da *thread* for interrompida. No caso a chamada a `provider` resultar numa excepção: (a) a chamada a `Value` nessa *thread* deve resultar no lançamento dessa excepção; (b) se o número de tentativas ainda não tiver sido excedido e existirem *threads* em espera, deve ser seleccionada a mais antiga para a retentativa do cálculo através da função `provider`; (c) quando o número de tentativas é excedido, todas as *threads* à espera na propriedade `Value`, ou que chamem essa propriedade no futuro, devem retornar lançando excepção `InvalidOperationException`.

4. Implemente, em *Java* ou C#, com base nos monitores implícitos ou explícitos (*Java*), a classe `RwSemaphore` que implementa um sincronizador com uma semântica idêntica à do *read/write semaphore* disponível no *kernel* do sistema operativo *Linux*. A interface pública desta classe, definida em C#, é a seguinte:

```
public class RwSemaphore {
  public void DownRead();            // throws ThreadInterruptedException
  public void DownWrite();           // throws ThreadInterruptedException
  public void UpRead();              // throws InvalidOperationException
  public void UpWrite();             // throws InvalidOperationException
  public void DowngradeWriter();     // throws InvalidOperationException
}
```

Os métodos `DownRead` e `DownWrite` adquirem a posse do semáforo para leitura ou escrita, respectivamente. Os métodos `UpRead` e `UpWrite` libertam o semáforo depois do mesmo ter sido adquirido para leitura ou escrita, respectivamente. O método `DowngradeWriter`, que apenas pode ser invocado pelas *threads* que tenham adquirido o semáforo para escrita, liberta o acesso para escrita e, atomicamente, adquire acesso para leitura. Se o método `UpRead` for invocado por *threads* que não tenham previamente adquirido o semáforo para leitura, ou os métodos `UpWrite` e `DowngradeWriter` forem invocados por *threads* que não tenham previamente adquirido o semáforo para escrita, deve ser lançada a excepção `InvalidOperationException`. O acesso para leitura deve ser concedido às *threads* leitoras que se encontrem no início da fila de espera (ou de imediato, se a fila de espera estiver vazia quando é invocado o método `DownRead`) desde que o não tenha sido concedido acesso para escrita a outra *thread;* para ser concedido acesso para escrita à *thread* que se encontra à cabeça da fila (ou de imediato, se a fila estiver vazia quando é invocado o método `DownWrite`), é necessário que nenhuma outra *thread* tenha adquirido acesso para escrita ou para leitura.

Para que o semáforo seja equitativo na atribuição dos dois tipos de acesso (leitura e escrita), deverá ser utilizada uma única fila de espera, com disciplina FIFO, onde são inseridas por ordem de chegada as solicitações de aquisição pendentes. A implementação deve suportar o cancelamento dos métodos bloqueantes quando são interrompidas as *threads* bloqueadas (lançando `ThreadInterruptedException`) e deve optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

5. Implemente em *Java*, com base nos monitores implícitos ou explícitos, o sincronizador *synchronous thread pool executor*, que executa as funções que lhe são submetidas numa das *worker threads* que o sincronizador deve criar e gerir para esse efeito. A interface pública, definida em *Java*, da classe que implenenta este sincronizador é a seguinte:

```
public class SynchronousThreadPoolExecutor<T>{
  public SynchronousThreadPoolExecutor(int maxPoolSize, int keepAliveTime);
  public T execute(Callable<T> toCall) throws InterruptedException, Exception;
  public void shutdown();
}
//as defined in the java.util.concurrent package
Interface Callable<V> { public V call() throws Exception; }
```

O número máximo de *worker threads* (`maxPoolSize`) e o tempo máximo que uma *worker thread* pode estar inativa antes de terminar (`keepAliveTime`) são passados com argumentos para o construtor da classe `SynchronousThreadPoolExecutor`. A gestão, pelo sincronizador, das *worker threads* deve obedecer aos seguintes critérios: (a) se o número total de *worker threads* for inferior ao limite máximo especificado, é criada uma nova *worker thread* sempre que for submetida uma função para execução e não existir nenhuma *worker thread* disponível; (b) as *worker threads* deverão terminar após decorrerem mais do que `keepAliveTime` nanosegundos sem que sejam mobilizadas para executar uma função; (c) o número de *worker threads* existentes no *pool* em cada momento depende da atividade deste e pode variar entre zero e `maxPoolSize`.

As *threads* que pretendem executar funções através do *thread pool executor* invocam o método `execute`, especificando a função a executar através do parâmetro `toCall`. Este método bloqueia sempre a *thread*

invocante até que uma das *worker threads* conclua a execução da função especificada, e pode terminar: (a) normalmente, devolvendo a instância do tipo `T` devolvida pela função, ou; (b) excepcionalmente, lançando a mesma excepção que foi lançada aquando da chamada à função. Até ao momento em que a *thread* dedicada considerar uma função para execução, é possível interromper a execução do método execute; contudo, se a interrupção ocorrer depois da função ser aceite para execução, o método execute deve ser processado normalmente, sendo a interrupção memorizada de forma a que possa vir a ser lançada pela *thread* mais tarde.

A chamada ao método `shutdown` coloca o executor em modo *shutdown*. Neste modo, todas as chamadas ao método execute deverão lançar a excepção `IllegalStateException`. Contudo, todas as submissões para execução feitas antes da chamada ao método `shutdown` devem ser processadas normalmente. O método `shutdown` deverá bloquear a *thread* invocante até que sejam executados todos os itens de trabalho aceites pelo executor e que tenham terminado todas as *worker threads* ativas.

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

Data limite de entrega: 14 de Maio de 201

ISEL, 5 de Abril de 2017