Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Verão de 2019/2020, Série de Exercícios 1

Escreva classes *thread-safe* para realizar os sincronizadores especificados utilizando os monitores implícitos da plataforma .NET, a extensão aos monitores do .NET, 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 o sincronizador *bounded lazy*, para suportar a computação de valores apenas quando são necessários. A interface pública deste sincronizador, em *Java*, é a seguinte:

   ```
   public class BoundedLazy<E> {
     public BoundedLazy(Supplier<E> supplier, int lives);
     public Optional<E> get(long timeout) throws InterruptedException;
   }
   ```

   A chamada do método **`get`** deve ter o seguinte comportamento: (a) caso o valor já tenha sido calculado e ainda não tenha sido usado **`lives`** vezes, retorna esse valor; (b) caso o valor ainda não tenha sido calculado ou já tenha sido usado **`lives`** vezes, inicia o cálculo de novo valor, chamando **`supplier`** na própria *thread* invocante (depois de sair do monitor) e retorna o valor resultante; (c) caso já existe outra *thread* a realizar esse cálculo, espera até que o valor esteja calculado; (d) retorna *empty* caso o tempo de espera exceda **`timeout`**, e; (e) lança **`InterruptedException`** se a espera da *thread* for interrompida. Caso a chamada a **`supplier`** resulte numa excepção, o objeto passa para um estado de erro, lançando essa excepção em todas as chamadas a **`get`**.

2. Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *brodcast box* cuja interface pública, em *Java*, é a seguinte:

   ```
   public class BroadcastBox<E> {
     public int deliverToAll(E message);
     public Optional<E> receive(long timeout) throws InterruptedException;
   }
   ```

   O método **`deliverToAll`** entrega uma mensagem a todas as *threads* à espera de receber mensagem nesse momento e nunca bloqueia a *thread* invocante. Este método retorna o número exacto de *threads* que receberam a mensagem; se não existem *threads* bloqueadas a mensagem é descartada e o método retorna 0. O método **`receive`** permite receber uma mensagem, e termina: (a) com sucesso, retornado um **`Optional`** com a mensagem recebida; (b) retornando **`Optional.empty()`** se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`InterruptedException`** quando a espera da *thread* é interrompida.

3. Implemente o sincronizador *exchanger*, cuja interface pública, em *Java*, é a seguinte:

   ```
   public class Exchanger<T> {
     public Optional<T> exchange(T mydata, long timeout) throws InterruptedException;
   }
   ```

   Este sincronizador suporta a troca de informação entre pares de *threads*. As *threads* que utilizam este sincronizador manifestam a sua disponibilidade para iniciar uma troca invocando o método **`exchange`**, especificando o objecto que pretendem entregar à *thread* parceira (**`mydata`**) e, opcionalmente, o tempo limite da espera pela realização da troca (**`timeout`**). O método **`exchange`** termina: (a) devolvendo um *optional* com valor, quando é realizada a troca com outra *thread*, sendo o objecto por ela oferecido retornado no valor desse *optional*; (b) devolvendo um *optional* vazio, se expirar o limite do tempo de espera especificado, ou; (c) lançando **`InterruptedException`** quando a espera da *thread* for interrompida.

4. Implemente o sincronizador *transfer queue*, para suportar a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico **`E`**. A interface pública deste sincronizador, em *Java*, é a seguinte:

   ```
   public class TransferQueue<E> {
     public void put(E message);
     public boolean transfer(E message, long timeout) throws InterruptedException;
   ```

```
  public E take(int timeout) throws InterruptedException;
}
```

O método **put** entrega uma mensagem à fila e nunca bloqueia a *thread* invocante. O método **transfer** entrega uma mensagem à fila, com garantia da respectiva recepção, pelo que pode bloquear a *thread* invocante até que a mensagem seja recebida por uma *thread* consumidora, e termina: (a) com sucesso, retornando **true** quando a mensagem é recebida; (b) retornando **false** se for excedido o limite especificado para o tempo de espera, e; (c) lançando **InterruptedException** quando a espera da *thread* é Interrompida. Quando o método **transfer** termina sem sucesso, a respectiva mensagem deve ser descartada.

O método **take** recebe a próxima mensagem da fila, e termina: (a) com sucesso, retornando a referência para a mensagem recebida; (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 implementação do sincronizador deve-se optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

5. [Opcional] Implemente o sincronizador *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, em *Java*, é a seguinte:

```
public class ThreadPoolExecutor {
  public ThreadPoolExecutor(int maxPoolSize, int keepAliveTime);
  public <T> Result<T> execute(Callable<T> command) throws InterruptedException;
  public void shutdown();
  public boolean awaitTermination(int timeout) throws InterruptedException;

}

public interface Result<T> {
  boolean isComplete();
  boolean tryCancel();
  Optional<T> get(int timeout) throws Exception;
}
```

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 **ThreadPoolExecutor**. A gestão, pelo 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**. O método **execute** retorna imediatamente um valor do tipo **Result<T>**, com a seguinte caracterização. O método **isComplete** retorna **true** se o comando já foi executado e **false** em caso contrário. O método **tryCancel** tenta cancelar a execução do comando, retornando **true** apenas se esse cancelamento foi possível. O método **get** é usado para obter o valor do comando, e termina: (a) normalmente, devolvendo o valor do comando, se o comando foi executado com sucesso; (b) excepcionalmente, lançando a excepção **CancellationException**, se a execução do comando foi cancelada; (c) excepcionalmente, devolvendo *empty*, se expirar o limite de tempo especificado com **timeout** sem que o comando seja concluído, ou; (d) excepcionalmente, lançando **InterruptedException**, se o bloqueio da *thread* for interrompido. O método **get** pode também lançar a excepção lançada aquando da execução do **Callable**.

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: 3 de Maio de 2020

ISEL, 3 de Abril de 2020