Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Verão de 2021/2022, Primeira Série de Exercícios

Realize classes *thread-safe* com a implementação dos seguintes sincronizadores. Para cada sincronizador, apresente pelo menos um dos programas ou testes que utilizou para verificar a correção da respectiva implementação. A resolução deve também conter documentação, na forma de comentários no ficheiro fonte, incluindo:

- Técnica usada (e.g. *monitor-style* vs delegação de execução/*kernel-style*).
- Aspectos de implementação não óbvios.

A entrega deve ser feita através da criação da *tag* 0.1.0 no repositório individual de cada aluno.

1. Implemente o sincronizador **`Exchanger`** com funcionalidade semelhante ao sincronizador com o mesmo nome presente na biblioteca standard do Java.

```
class Exchanger<T> {
   @Throws(InterruptedException::class)
   fun exchange(value: T, timeout: Duration): T? { … }
}
```

   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 (**`value`**) e a duração limite da espera pela realização da troca (**`timeout`**). O método **`exchange`** termina: (a) devolvendo o valor trocado, quando é realizada a troca com outra *thread*; (b) devolvendo **`null`**, se expirar o limite do tempo de espera especificado, ou; (c) lançando **`InterruptedException`** quando a espera da *thread* for interrompida.

   Se a chamada C1 retornou o valor submetido pela chamada C2, então a chamada C2 retornou obrigatoriamente o valor submetido pela chamada C1.

2. Implemente o sincronizador *blocking message queue*, para suportar a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico **`T`**. A comunicação deve usar o critério FIFO (*first in first out*): dadas duas mensagens colocadas na fila, a primeira a ser entregue a um consumidor deve ser a primeira que foi colocada na fila; caso existam dois ou mais consumidores à espera de uma mensagem, o primeiro a ver o seu pedido satisfeito é o que está à espera há mais tempo. O número máximo de elementos presentes na fila é determinado pelo parâmetro **`capacity`**, definido no construtor.

   A interface pública deste sincronizador é a seguinte:

```
class BlockingMessageQueue<T>(private val capacity: Int) {
   @Throws(InterruptedException::class)
   fun tryEnqueue(messages: List<T>, timeout: Duration): Boolean { … }
   @Throws(InterruptedException::class)
   fun tryDequeue(timeout: Duration): T? { … }
}
```

O método **tryEnqueue** entrega uma lista de mensagens à fila, ficando bloqueado caso a fila não tenha capacidade disponível para essas mensagens. Esse bloqueio deve terminar mal todas as mensagens possam ser colocadas na fila sem exceder a sua capacidade. Caso o tempo definido seja ultrapassado sem que todas as mensagens possam ser colocadas na fila, o método deve retornar **false**. Note-se que este método não tem de esperar que as mensagens sejam entregues a um consumidor; apenas que possam ser colocadas na fila. O retorno do método indica se todas as mensagens foram colocadas na fila (**true**) ou se nenhuma mensagem foi colocada na fila (**false**).

O método **tryDequeue** remove e retorna um elemento da fila, ficando bloqueado caso a fila esteja vazia. O bloqueio é limitado pela duração definida por **timeout**. Caso este tempo seja ultrapassado o método deve retornar **null**.

3. Implemente o sincronizador *thread pool executor*, que executa cada comando que lhe é submetido numa das *worker threads* que o sincronizador cria e gere para o efeito. A interface pública deste sincronizador é a seguinte:

```
class ThreadPoolExecutor(
   private val maxThreadPoolSize: Int,
   private val keepAliveTime: Duration,
) {
   @Throws(RejectedExecutionException::class)
   fun execute(runnable: Runnable): Unit { … }
   fun shutdown(): Unit { … }
   @Throws(InterruptedException::class)
   fun awaitTermination(timeout: Duration): Boolean { … }
}
```

O número máximo de *worker threads* (**maxThreadPoolSize**) e o tempo máximo que uma *worker thread* pode estar inactiva antes de terminar (**keepAliveTime**) são passados como 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 *runnable* para execução e não existir nenhuma *worker thread* disponível; (2) as *worker threads* deverão terminar após decorrer o tempo especificado em **keepAliveTime** 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 **maxThreadPoolSize**.

As *threads* que pretendem executar funções através do *thread pool executor* invocam o método **execute**, especificando o comando a executar com o argumento **runnable**. Este método retorna imediatamente.

A chamada ao método **`shutdown`** coloca o executor em modo de encerramento 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 a qualquer *thread* invocante sincronizar-se com a conclusão do processo de encerramento do executor, isto é, aguarda até que sejam executados todos os comandos aceites e que todas as *worker threads* activas terminem, e pode acabar: (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 encerramento termine, ou; (c) excepcionalmente, lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido.

4. Realize a variação do sincronizador *blocking message queue* em que o método **`tryDequeue`** inicia a remoção de uma mensagem da fila, retornando imediatamente um *representante* dessa operação, implementando a interface **`Future<T>`** e *thread-safe*. Para a resolução deste exercício não utilize implementações de **`Future<T>`** existentes na biblioteca de classes da plataforma Java. Todos os métodos potencialmente bloqueantes das implementações de **`Future<T>`** devem ser sensíveis a interrupções, tratando-as de acordo com o protocolo definido na plataforma Java.

Data limite de entrega: 23 de abril de 2022

ISEL, 28 de março de 2022