Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Verão de 2022/2023, 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 **`NAryExchanger`** com funcionalidade semelhante ao sincronizador **`Exchanger`** com o mesmo nome presente na biblioteca standard do Java, mas generalizado para dimensões maiores que 2.

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

Este sincronizador suporta a troca de informação entre grupos de *threads* com dimensão **`groupSize`**, em que **`groupSize`** é maior ou igual a dois. As *threads* que utilizam este sincronizador manifestam a sua disponibilidade para iniciar uma troca invocando o método **`exchange`**, especificando o objecto que pretendem entregar ao grupo (**`value`**) e a duração limite da espera pela realização da troca (**`timeout`**). O método **`exchange`** termina: (a) devolvendo os valores disponibilizados por todas a threads do grupo; (b) devolvendo **`null`**, se expirar o limite do tempo de espera especificado, ou; (c) lançando **`InterruptedException`** quando a espera da *thread* for interrompida.

Um grupo é formado e a troca é realizada sempre que existirem **`groupSize`** chamadas em espera. Em adição:

a) Se a chamada *C* retornou a lista *L*, então a dimensão de *L* é **`groupSize`**.

b) Se a chamada *C* retornou a lista *L*, então o valor entregue na chamada *C* está presente na lista *L*.

c) Se a chamada C_1 retornou a lista *L*, contendo os valores entregues pelas chamadas C_1, … C_N, então a chamada C_i retornou também a lista *L*, para $1 \leq i \leq N$.

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 entregue à 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 armazenados 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(message: T, timeout: Duration): Boolean { … }
   @Throws(InterruptedException::class)
   fun tryDequeue(nOfMessages: Int, timeout: Duration): List<T>? { … }
}
```

   O método **`tryEnqueue`** entrega uma mensagem à fila, ficando bloqueado caso a fila não tenha capacidade disponível para essa mensagem. Esse bloqueio deve terminar quando a mensagem possa ser colocada na fila sem exceder a sua capacidade, ou entregue a um consumidor. Caso o tempo definido seja ultrapassado, o método deve retornar **`false`**.

   O método **`tryDequeue`** remove e retorna uma lista de mensagens da fila, com dimensão **`nOfMessages`**, ficando bloqueado enquanto o pedido não puder ser satisfeito por completo. O bloqueio é limitado pela duração definida por **`timeout`**. Caso este tempo seja ultrapassado o método deve retornar **`null`**.

   Ambos os métodos devem ser sensíveis a interrupções, tratando-as de acordo com o protocolo definido na plataforma Java.

3. Implemente o sincronizador *thread pool executor*, em que cada comando submetido é executado 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 das *worker threads pelo sincronizador* 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. Acrescente ao *thread pool executor* realizado na questão 3. o seguinte método

```
fun <T> execute(callable: Callable<T>): Future<T> { … }
```

que agenda a execução de um *callable* e retorna 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: 16 de abril de 2023

ISEL, 20 de março de 2023