Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Verão de 2023/2024, 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 de cada grupo..

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

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

3. 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(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 essa lista. Esse bloqueio deve terminar quando a lista de mensagens puder ser colocada na fila sem exceder a sua capacidade, ou entregue a consumidores. Caso o tempo definido seja ultrapassado, o método deve retornar **`false`**.

   O método **`tryDequeue`** remove e retorna uma mensagem da fila, 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.

4. 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.

5. Realize a função

```
@Throws(InterruptedException::class)
fun <T> race(suppliers: List<()->T>, timeout: Duration): T?
```

Esta função recebe uma lista de *suppliers* (funções que produzem um valor do tipo **`T`**) e promove a sua execução em *threads* virtuais diferentes. A produção de um valor por uma destas funções deve cancelar a execução das restantes funções, usando o mecanismo de interrupções. A chamada da função deve apenas

acabar quando todas as *threads* criadas internamente tiverem acabado, retornando o primeiro valor a ter sido produzido por um dos *suppliers*.

Caso tenha decorrido a duração **`timeout`** desde o início da chamada e nenhum valor tiver sido produzido, então deve ser tentado o cancelamento de todas as funções. O mesmo deve acontecer caso a *thread* onde a função **`race`** foi executada seja alvo de uma interrupção. Em ambos os casos, a função **`race`** apenas acaba quando todas as *threads* criadas internamente tiverem terminado.

A resolução desta questão não deve usar funções pertencentes à API (*Application Programming Interface*) de Concorrência Estruturada (JEP 453).

Data limite de entrega: 2 de abril de 2024

ISEL, 11 de março de 2024