Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Inverno 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 *message box*, com os métodos apresentados em seguida.

```
public class MessageBox<T> {
   public Optional<T> waitForMessage(long timeout) throws InterruptedException;
   public int sendToAll(T message);
}
```

O método **`waitForMessage`** bloqueia a *thread* invocante até que uma mensagem seja enviada através do método **`sendToAll`**. O método **`waitForMessage`** pode terminar com: 1) um objecto **`Optional`** contendo a mensagem enviada; 2) um objecto **`Optional`** vazio, caso o tempo de espera definido por *timeout* seja excedido sem que uma mensagem seja enviada; 3) com o lançamento duma excepção do tipo **`InterruptedException`**, caso a *thread* seja interrompida enquanto em espera.

O método **`sendToAll`** deve retornar o número exacto de *threads* que receberam a mensagem, podendo este valor ser zero (não existiam *threads* à espera de mensagem), um, ou maior que um. A mensagem passada na chamada **`sendToAll`** não deve ficar disponível para chamadas futuras do método **`waitForMessage`**.

2. Realize o sincronizador **`SemaphoreWithShutdown`**, que representa um semáforo com aquisição e libertação unária, com garantia de ordem FIFO na atribuição de unidades e com a interface apresentada em seguida

```
public class SemaphoreWithShutdown {
  public SemaphoreWithShutdown(int initialUnits);
  public boolean acquireSingle(long timeout)
                                throws InterruptedException, CancellationException;
  public void releaseSingle();
  public void startShutdown();
  public boolean waitShutdownCompleted(long timeout) throws InterruptedException;
}s
```

O método **`startShutdown`** coloca o semáforo num estado de encerramento. Nesse estado todas as chamadas a **`acquireSingle`**, futuras ou atualmente pendentes, devem terminar com o lançamento da excepção **`CancellationException`**. O processo de encerramento é considerado completo quando as unidades

disponíveis no semáforo forem iguais ao valor inicial, definido na construção. Chamadas ao método **`waitShutdownCompleted`** esperam que o processo de encerramento esteja concluído.

Os métodos **`acquireSingle`** e **`waitShutdownCompleted`** recebem o valor do tempo máximo de espera, retornando **`false`** se e só se o fim da sua execução se dever à expiração desse tempo. Ambos os métodos devem ser sensíveis a interrupções, tratando-as de acordo com o protocolo do Java para métodos potencialmente bloqueantes.

3. Implemente o sincronizador *message queue*, para suportar a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico **`E`**. 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, em *Java*, é a seguinte:

```
public class BlockingMessageQueue<E> {

  public BlockingMessageQueue(int capacity);
  public boolean enqueue(E message, long timeout) throws InterruptedException;
  public Future<E> dequeue();
}
```

   O método **`enqueue`** entrega uma mensagem à fila, ficando bloqueado caso a fila esteja cheia. Esse bloqueio deve terminar mal a mensagem possa ser colocada na fila sem exceder a sua capacidade. Caso o tempo definido seja ultrapassado sem que a mensagem possa ser colocada na fila, o método deve retornar **`false`**. Note-se que este método não tem de esperar que a mensagem seja entregue a um consumidor; apenas que possa ser colocada na fila.

   O método **`dequeue`** inicia a remoção de uma mensagem da fila, retornando um *representante* dessa operação, implementando a interface **`Future<E>`** e *thread-safe*. Para a resolução deste exercício não utilize implementações de **`Future<E>`** existentes na biblioteca de classes do Java. Todos os métodos potencialmente bloqueantes devem ser sensíveis a interrupções, tratando-as de acordo com o protocolo definido no Java.

4. Implemente o sincronizador *keyed 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 KeyedThreadPoolExecutor {
  public KeyedThreadPoolExecutor (int maxPoolSize, int keepAliveTime);
  public void execute(Runnable runnable, Object key);
  public void shutdown();
  public boolean awaitTermination(int timeout) throws InterruptedException;

}
```

O número máximo de *worker threads* (**maxPoolSize**) e o tempo máximo que uma *worker thread* pode estar inactiva antes de terminar (**keepAliveTime**) são passados como argumentos para o construtor da classe **KeyedThreadPoolExecutor**. 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 **runnable** (o parâmetro **key** é descrito posteriormente). 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, 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.

Este executor deve também garantir que nunca está em execução simultânea mais do que um *runnable* associado à mesma chave (usando **equals** como critério de igualdade), independentemente de existirem *worker threads* disponíveis.

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

Data limite de entrega: 6 de novembro de 2021

ISEL, 18 de outubro de 2021