Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Verão de 2020/2021, Primeira Série de Exercícios

Escreva classes *thread-safe* para realizar os sincronizadores especificados, utilizando os monitores implícitos da plataforma .NET, a extensão aos monitores do .NET, ou os monitores disponíveis na plataforma *Java*. Para cada sincronizador, apresente pelo menos um dos programas ou testes que utilizou para verificar a correção da respectiva implementação. A entrega deve ser feita através da criação da tag **`0.1.0`** no repositório individual de cada aluno.

1. Realize o sincronizador **`SemaphoreWithShutdown`**, que representa um semáforo com aquisição e libertação unária, sem garantia de ordem 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;
}
```

   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. Minimize o número de objetos alocados durante a operação do semáforo, bem como as comutações de contexto.

2. 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 mensage, o primeiro a ver o seu pedido satisfeito é o que está à espera há mais tempo.. A interface pública deste sincronizador, em *Java*, é a seguinte:

```
public class MessageQueue<E> {
  public DeliveryStatus enqueue(E message);
  public Optional<E> dequeue(long timeout) throws InterruptedException;
}
```

```
public interface DeliveryStatus {
  boolean isDelivered();
  boolean isCancelled();
  boolean tryCancel();
  boolean awaitComplete(long timeout) throws InterruptedException;
}
```

   O método **`enqueue`** entrega uma mensagem à fila e nunca bloqueia a *thread* invocante. O método **`dequeue`** retira uma mensagem da fila, pelo que pode bloquear a *thread* invocante até que uma mensagem esteja disponível, e termina: (a) com sucesso, retornando um **`Optional`** com a mensagem removida; (b) retornando um **`Optional`** vazio se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`InterruptedException`** quando a espera da *thread* é Interrompida.

O método **enqueue** retorna um objeto para verificar e sincronizar com o estado da entrega, possuindo métodos para:

- Verificar se a mensagem já foi retirada da fila através do método **dequeue**.
- Verificar se a inserção foi cancelada, i.e., a mensagem garantidamente não vai ser usada para satisfazer uma operação de **dequeue**.
- Tentar cancelar a inserção. Este método deve retornar **true** apenas quando o cancelamento foi realizado com sucesso, o que nem sempre é possível.
- Bloquear a *thread* invocadora até que a mensagem seja retirada da fila através do método **dequeue**, a inserção seja cancelada, ou o tempo definido para *timeout* seja ultrapassado.

O objeto retornado por uma chamada a **enqueue** deve poder ser usado por múltiplas *threads* em simultâneo. Por exemplo, uma *thread* pode chamar **tryCancel** enquanto outra *thread* está bloqueada no método **awaitComplete**.

Todos os métodos potencialmente bloqueantes podem lançar **InterruptedException** e recebem o tempo máximo de espera em milisegundos.

3. 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 void execute(Runnable runnable);
  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 **runnabl**). Este método retorna imediatamente.
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 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 *shutdown* termine, ou; (c) excepcionalmente, lançando **InterruptedException**, se o bloqueio da *thread* for interrompido.

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

Data limite de entrega: 24 de abril de 2021

ISEL, 20 de março de 2021