Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Inverno de 2018/2019

Série de Exercícios 1

Escreva classes *thread-safe* para realizar os sincronizadores especificados utilizando os monitores implícitos da plataforma .NET e/ou os monitores implícitos ou os monitores explícitos disponíveis no *Java*. Para cada sincronizador, apresente pelo menos um dos programas ou testes que utilizou para verificar a correção da respectiva implementação.

1. Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *keyed exchanger*, cuja interface pública é a seguinte:

   ```
   public class KeyedExchanger<T> {
     public Optional<T> exchange(int ky, T mydata, int timeout) throws InterruptedException;
   }
   ```

   Este sincronizador suporta a troca de informação entre pares de *threads* identificados por uma chave. As *threads* que utilizam este sincronizador manifestam a sua disponibilidade para iniciar uma troca invocando o método **`exchange`**, especificando a identificação do par (**`key`**), o objecto que pretendem entregar à *thread* parceira (**`mydata`**) e, opcionalmente, o tempo limite da espera pela troca (**`timeout`**).
   O método **`exchange`** termina: devolvendo um *optional* com valor, quando é realizada a troca com outra *thread*, sendo o objecto por ela oferecido retornado no valor desse *optional*; devolvendo um *optional* vazio, se expirar o limite do tempo de espera especificado, ou; lançando **`ThreadInterruptedException`** quando a espera da *thread* for interrompida.

2. Implemente em C# ou *Java* a classe **`EventBus`** com a seguinte interface pública:

   ```
   public class EventBus {
     public EventBus(int maxPending);
     public void SubscribeEvent<T>(Action<T> handler) where T: class;
     public void PublishEvent<E>(E message) where E: class;
     public void Shutdown();
   }
   ```

Esta classe visa disponibilizar um mecanismo para que eventos dum sistema (e.g., utilizador registrou-se com sucesso, ligação à base de dados falhou) sejam publicados para todos os subscritores interessados nesse tipo de evento. Por exemplo o sistema pode ter um componente subscrito no evento de utilizador registrado com sucesso de forma a enviar-lhe um email de boas vindas. A publicação dum evento consiste no envio de um objecto, designado por mensagem, para todos os subscritores registados no tipo desse objecto. Tipos diferentes de eventos são representados por tipos .NET diferentes.
O método **`SubscribeEvent`** regista um *handler* para ser executado sempre que for publicada uma mensagem do tipo **`T`**, sendo o handler executado pela *thread* que procedeu ao respectivo registo com **`SubscribeEvent`**. Note-se que a chamada a este método é bloqueante, só retornando após um *shutdown* ou se a respectiva *thread* for interrompida. O método **`PublishEvent`**, que nunca bloqueia a *thread* invocante, envia a mensagem especificada para o *bus* de modo a que esta seja processada por todos os *handlers* registados até ao momento, independentemente de estarem, ou não, a processar outra mensagem. Caso existam mais do que **`maxPending`** mensagens para serem processadas pelo mesmo *handler*, o método **`PublishEvent`** deve descartar o evento para esse *handler*.
O método **`SubscribeEvent`** deve lançar **`ThreadInterruptedException`** no caso da *thread* invocante ser interrompida enquanto estiver bloqueada ou durante a execução do *handler*. Após a chamada ao método **`Shutdown`**, posteriores chamadas ao método **`PublishEvent`** devem lançar **`InvalidOperationException`** e todas as chamadas ao método **`SubscribeEvent`** deverão retornar após serem processadas todas as mensagens publicadas. O método **`Shutdown`** deve bloquear a *thread* invocante até que o processo de *shutdown* esteja concluído, isto é, tenha sido completado o processamento de todos as mensagens aceites pelo *bus*.

3. Implemente em *Java* ou C#, como base nos monitores implícitos ou explícitos, o sincronizador *message queue*, cuja interface pública em Java é a seguinte:

```
public class MessageQueue<T> {
  public SendStatus send(T sentMsg);
  public Optional<T> receive(int timeout) throws InterruptedException;
}
public interface SendStatus {
  boolean isSent();
  boolean tryCancel();
  boolean await(int timeout) throws InterruptedException;
}
```

Este sincronizador permite a comunicação entre *threads* produtoras e *threads* consumidoras. A operação **`send`** entrega uma mensagem à fila (**`sentMsg`**), e termina imediatamente retornando um objecto que implementa a interface **`SendStatus`**. Este objecto permite a sincronização com a entrega da respectiva mensagem a outra *thread*. O método **`isSent`** retorna **`true`** se a mensagem já foi entregue a outra *thread*, **`false`** em caso contrário.
O método **`await`** sincroniza com a entrega da mensagem: (a) devolvendo **`true`** quando a mensagem for recebida por uma *thread* consumidora; (b) devolvendo **`false`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido.
O método **`tryCancel`** tenta remover a mensagem da fila, retornando o sucesso dessa remoção (a remoção pode já não ser possível).
O método **`receive`** permite receber uma mensagem da fila, e termina: (a) devolvendo um *optional* com a mensagem, em caso de sucesso; (b) devolvendo um *optional* vazio se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido.
A implementação do sincronizador deve optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

4. Implemente em *Java*, com base nos monitores implícitos ou explícitos, o sincronizador *simple 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 é a seguinte:

```
public class SimpleThreadPoolExecutor {
  public SimpleThreadPoolExecutor(int maxPoolSize, int keepAliveTime);
  public boolean execute(Runnable command, int timeout) throws InterruptedException;
  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 com argumentos para o construtor da classe **`SimpleThreadPoolExecutor`**. 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 command. Este método pode bloquear a *thread* invocante, pois tem que garantir que o comando especificado foi entregue a uma *worker threads* para execução, e pode terminar: (a) normalmente, devolvendo true, se o comando foi entregue para execução; (b) excepcionalmente, lançando a excepção RejectedExecutionException, se o *thread pool* se encontrar em modo *shutting down*; (c) excepcionalmente, devolvendo false, se expirar o limite de tempo especificado com timeout sem que o comando seja entregue a uma *worker thread*, ou; (d) excepcionalmente, lançando InterruptedException, se o bloqueio da *thread* for interrompido.
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 terminar: (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 optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

Data limite de entrega: 24 de Outubro de 2018

ISEL, 26 de Setembro de 2018