Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Inverno de 2009/2010

Segunda série de exercícios

Escreva classes *thread-safe* para realizar os sincronizadores especificados, utilizando os monitores intrínsecos das plataformas de execução. Todos os sincronizadores a implementar suportam cancelamento e desistência. Para cada sincronizador, implemente um conjunto de métodos de teste para verificar se a implementação cumpre a respectiva especificação (ver Cap. 12 do livro "Java Concurrency in Practice").

1. Implemente, em C# 2.0, o sincronizador `FutureHolder<Tval>`, cujas instâncias representam promessas de valores do tipo `Tval`. O sincronizador fornece as operações `Set` e `Get` usadas para estabelecer e obter, respectivamente, o valor mantido pelo sincronizador. As chamadas à operação `Get` são bloqueantes até que o valor seja estabelecido por via de chamada a `Set`.

2. [opcional] Implemente, em C# 2.0, a classe `Cache<Tkey, Tval>`, cujas instâncias representam *caches thread-safe*. Cada *cache* armazena valores (instâncias de `Tval`) produzidos a partir chaves (instâncias de `Tkey`). A especificação do procedimento para cálculo do valor associado a uma dada chave é realizada através da instância de `delegate Tval Func<Tkey, Tval>(Tkey key)` passada como parâmetro de construção. A classe fornece a operação `public Tval Get(Tkey)` que retorna o valor que corresponde à chave recebida, promovendo, se necessário, o cálculo e respectivo armazenamento para solicitações posteriores. A implementação a apresentar suporta o aumento de escala nos acessos concorrentes e promove a reciclagem das entradas da *cache* cujos valores não são solicitados há pelo menos T minutos (valor especificado como parâmetro de construção).

   Nota: Na implementação tenha em conta a discussão apresentada no Capítulo 5 do livro "*Java Concurrency in Practice"*.

3. Considere a classe `DocumentDB`, fornecida em anexo a este enunciado, escrita em Java para utilização em ambiente *single-threaded*. Escreva uma versão *thread-safe* da classe `DocumentDB` usando o mínimo de sincronização. No caso particular das secções críticas (blocos *synchronized* ou construções equivalentes), não deverá gastar mais do que uma, se chegar a ser necessário. Em conjunto com o código, apresente justificação para cada uma das suas decisões.

4. [opcional] Implemente em Java SE 6 ou C# 2.0 o sincronizador `PhasedGate`. A operação de `Wait(int timeout)` é bloqueante até que seja chamada pelas N *threads* participantes, cujo número é especificado no construtor. A última *thread* a invocar `Wait` não fica bloqueada, sendo libertadas as restantes. O sincronizador tem ainda a operação `RemoveParticipant`, que serve para remover uma unidade ao número de participantes.

5. Uma instância de `RendezvousPort` permite sincronizar um conjunto de *threads* clientes de um serviço com as *threads* servidoras desse serviço. As *threads* clientes realizam pedidos invocando o método `RequestService` passando, como argumento, um objecto que representa o pedido e ficando bloqueadas (em `RequestService`) até que seja produzida a respectiva resposta por uma das *threads* servidoras. As *threads* servidoras invocam `AcceptService` para esperarem por um pedido, recebendo, no retorno de `AcceptService`, o objecto passado como argumento a `RequestService` por uma das *threads* clientes. Quando o serviço estiver realizado, a resposta é entregue ao respectivo cliente invocando `CompleteService` passando como argumentos o objecto retornado pela chamada a `AcceptService` e a respectiva resposta. Nesse momento é desbloqueada a chamada a `RequestService` realizada pelo cliente, que retornará o objecto resposta entregue em `CompleteService`.

   5.1. Implemente, em Java SE 6 ou C# 2.0, o sincronizador `RendezvousPort`, procurando iniciar o atendimento aos pedidos dos clientes pela ordem de chegada a `RequestService`. A ordem de chegada das *threads* servidoras a `AcceptService` é irrelevante. O método `AcceptService` termina por *timeout* (lançando excepção do tipo `TimeoutException`) ou por cancelamento (lançando excepção do tipo `[Thread]InterruptedException`). Implemente apenas o suporte mínimo para a possibilidade de ocorrência de cancelamentos em `RequestService`.

5.2. Acrescente à implementação da alínea anterior suporte completo para abandono do método `RequestService` por *timeout* ou cancelamento, interrompendo a thread servidora, se esta já estiver a atender o pedido. Apresente ainda um padrão de escrita do código das *threads* servidoras que garanta a chamada a `CompleteService` para cada saída com sucesso de `AcceptService`. (Pode realizar ajustes na interface pública do sincronizador, se for necessário)

5.3. Implemente uma versão optimizada do sincronizador que minimize o número de comutações de contexto.

Data limite: 18 de Dezembro de 2009

Carlos Martins e Paulo Pereira
ISEL, 26 de Novembro de 2009