Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Inverno de 2011/2012

Primeira série de exercícios

Observações:

- Para cada exercício inclua o programa de teste e os recursos necessários à geração do respectivo executável (i.e. soluções e projectos do IDE usado, ficheiros de comandos ou ficheiros com o texto explicativo do processo de construção).
- Escreva classes *thread-safe* para realizar os sincronizadores especificados utilizando os monitores intrínsecos das plataformas CLI e Java. Todas as implementações devem prever a interrupção das *threads* bloqueadas na variável condição do respectivo monitor e todas as operações bloqueantes devem permitir especificar um limite para o tempo em que a *thread* corrente pode ficar bloqueada (*timeout*).

1. Realize os programas para determinar de forma aproximada o tempo necessário à comutação entre *threads* nas seguintes plataformas:
   - No sistema operativo Windows, usando a linguagem de programação C e a API Windows.
   - Na plataforma .NET, usando a linguagem C# e a CLI.

   Apresente os resultados em tempo absoluto e em número de ciclos de relógio do processador, indicando também as características do computador usado.

2. Implemente em Java o sincronizador *phased gate*, com base na classe `PhasedGate` que define a operação de `wait`. Esta operação é bloqueante até que seja chamada pelas N *threads* participantes (o valor de N 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. Note que as instâncias de `PhasedGate` são de utilização única.

3. Implemente em Java o sincronizador *transient signal*, com base na classe `TransientSignal` que define as operações `await`, `signalOne` e `signalAll`. A chamada a `await` é SEMPRE bloqueante, retornando `true` quando tiver ocorrido uma sinalização explícita (via `signalOne` ou `signalAll`) ou `false` se ocorrer *timeout*. A invocação de `signalOne` liberta uma das *threads* bloqueadas, se existir alguma. A invocação de `signalAll` liberta todas as *threads* bloqueadas nesse momento. Caso não existam *threads* bloqueadas, as chamadas a `signalOne` ou `signalAll` não produzem efeito. Na implementação tenha em consideração que não é necessário desbloquear as *threads* por ordem de chegada.

4. Implemente em C# o sincronizador *exchanger* `Exchanger<T>` que permite a troca, entre pares de *threads*, de mensagens definidas por instâncias do tipo `T`. A classe que implementa o sincronizador deve definir, pelo menos, o seguinte método:
   - Método `bool Exchange(T mine, int timeout, out T yours)`, que é chamado pelas *threads* para oferecer uma mensagem (parâmetro `mine`) e receber a mensagem oferecida pela *thread* com que emparelham (parâmetro `yours`). Quando a troca de mensagens não pode ser realizada de imediato (não existe já uma *thread* bloqueada), a *thread* corrente fica bloqueada até que outra *thread* invoque o método `Exchange`, seja interrompida ou expire o limite de tempo, especificado através do parâmetro `timeout`.

5. Implemente em C# o sincronizador *rendezvous channel*, com base na classe genérica `RendezvousChannel<S,R>`. O *rendezvous channel* serve para sincronizar a comunicação entre *threads* cliente e *threads* servidoras, com a seguinte semântica:
   - As *threads* cliente realizam pedidos de serviço invocando o método `bool Request(S service, int timeout, out R response)`. O objecto, do tipo `S`, passado através do parâmetro `service` descreve o pedido de serviço. Se o serviço for executado com sucesso por

uma *thread* servidora, este método devolve `true` e a resposta ao pedido de serviço, um objecto do tipo `R`, é passada através do parâmetro `response`; se pedido de serviço não for aceite de imediato, por não existir nenhuma *thread* servidora disponível, a *thread* cliente fica bloqueada até que o pedido de serviço seja aceite, a *thread* cliente seja interrompida ou expire o limite de tempo especificado através do parâmetro `timeout`. (Quando existe desistência o método `Request` devolve `false`.) Dado que não está prevista nenhuma forma de interromper o processamento de um pedido de serviço já aceite por uma *thread* servidora, as *threads* cliente não poderão desistir, devido a interrupção ou *timeout*, após terem iniciado o *rendezvous* com uma *thread* servidora, devendo esperar incondicionalmente que o serviço seja concluído, isto é, a *thread* servidora invoque o método `Reply`, com o respectivo *rendezvous token*.

- Sempre que uma *thread* servidora estiver em condições de processar pedidos de serviço, invoca o método `object Accept (int timeout, out S service)`. Quando um pedido de serviço é aceite, a descrição do pedido de serviço é passado através do parâmetro de saída `service` e o método `Accept` devolve também um *rendezvous token* (i.e., um objecto opaco, cujo tipo é definido pela implementação) para identificar um *rendezvous* particular. Quando não existe nenhum pedido de serviço pendente, a *thread* servidora fica bloqueada até que seja solicitado um pedido de serviço, seja interrompida ou expire o limite de tempo especificado através do parâmetro `timeout`. (Este método deve devolver `null` como *rendezvous token* para indicar que a *thread* servidora retornou por desistência.)
- Quando uma *thread* servidora quer indicar a conclusão de um serviço particular (definido pelo respectivo *rendezvous token*) e devolver o respectivo resultado, invoca o método `void Reply(object rendezVousToken, R response)`. Através do primeiro parâmetro é passada a identificação do *rendezvous* e através do parâmetro `response` o objecto do tipo `R`, que contém a resposta ao pedido de serviço.

Na resolução deste exercício procure minimizar as comutações de *thread*, usando as técnicas que foram discutidas nas aulas teóricas.

Data limite: 5 de Dezembro de 2011

Carlos Martins, Jorge Martins & Paulo Pereira
ISEL, 31 de Outubro de 2011