Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Inverno de 2011/12

Segunda série de exercícios

Parte I

1. Implemente o método estático `ParallelLoopResult For(int start, int end, ParallelOptions options ,Action<int> body)`, com a mesma semântica do método de idêntica assinatura da classe `System.Threading.Tasks.Parallel`. Tenha em atenção a eficiência da solução apresentada.

2. Explique o comportamento da execução do programa seguinte:

```
public static void Main() {
        Task t1 = Task.Factory.StartNew(() =>
        {
                Task.Factory.StartNew(() => {
                        throw new Exception("Child exception!");
                }, TaskCreationOptions.AttachedToParent);
                throw new Exception("Parent exception!");
        });
        ((IAsyncResult) t1).AsyncWaitHandle.WaitOne();
        t1 = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
        Console.ReadLine();
}
```

3. [Opcional] Implemente a classe `ThreadPerTaskScheduler`, especialização de `TaskScheduler`, que cria um mapeamento de um para um entre *tasks* e *threads*, isto é, executa cada *task* em *thread* distinta.

4. [Opcional] Considere a classe `Pipeline<TInput, TOutput>`, apresentada a seguir, que permite construir uma cadeia de transformadores. Cada passo da cadeia transforma uma sequência de elementos do tipo TInput em elementos do tipo `TOutput`. O tipo `TOutput` de um dado passo do `Pipeline` é o tipo `TInput` do passo seguinte. Cada passo do `Pipeline` é executado em *thread* distinta. A execução do método `Test` gera o *output*: `Impar Impar Par Par Impar Impar Par Par Impar Impar.` Na implementação da classe Pipeline considere a utilização da classe `BlockingCollection<T>.`

```
public class Pipeline<TInput, TOutput>  {

    public Pipeline(Func<TInput, TOutput> stage);

    /* Retorna um novo Pipeline ao qual foi acrescentado um passo que converte
      elementos do tipo TOutput em elementos do tipo TNextOutput */
    public Pipeline<TInput, TNextOutput>
    Next<TNextOutput>(Func<TOutput, TNextOutput> nextStage);

    /* executa o pipeline com possibilidade de cancelamento. O input é uma
      enumeração de elementos do tipo do primeiro passo e o output uma enumeração
      de elementos do tipo do último passo. */
    public IEnumerable<TOutput> Run(IEnumerable<TInput> _source, CancellationToken token);

     /* executa o pipeline sem possibilidade de cancelamento. */
     public IEnumerable<TOutput> Run(IEnumerable<TInput> _source);

     public  static void Test() {
          // sequência de inteiros de 1 a 10
          var source = 1.To(10);
          var p1 = new Pipeline<int, int>(i => i * 3)
                                .Next(i => i /2)
                                .Next(i => (i % 2 == 0) ? "Par" : "Impar");
          foreach (string i in p1.Run(source))
                          Console.WriteLine(i);

      }
}
```

Parte II

Acompanha este enunciado a implementação em C# de um servidor *single-threaded* que realiza *tracking* de ficheiros disponibilizados por máquinas que participam num sistema de partilha de ficheiros entre pares (e.g. *BitTorrent*). O servidor mantém informação relativa aos ficheiros (i.e., nome) existentes nos participantes e à sua localização (i.e., endereço IP e porto a serem usados para obter o ficheiro).

A comunicação entre o servidor e os clientes (i.e., participantes no sistema de partilha) é realizada através de protocolo proprietário baseado em pares pedido/resposta e sustentado no protocolo de transporte TCP. As ligações TCP são mantidas enquanto o cliente não solicita a sua terminação, existindo a possibilidade de vários pares pedido/resposta serem trocados usando a mesma ligação. O protocolo proprietário, documentado na implementação fornecida, oferece suporte para as seguintes operações: adição e remoção de localização de ficheiro; obtenção de lista de ficheiros, e; obtenção de lista de localizações de um dado ficheiro.

Os principais elementos da implementação fornecida são as classes `Listener` e `Handler`, cujas instâncias participam no serviço de pedidos. A classe `Store` mantém em memória volátil (por simplificação) a informação de *tracking*. A definição destas classes está acompanhada da respectiva documentação.

O servidor mantém registo das acções realizadas (classe `Logger`), que pode ser apresentado na consola ou em ficheiro.

1. Inspirando-se na estrutura do servidor *single-threaded* fornecido, implemente uma versão *multi-threaded* do servidor, com as seguintes características:
   - Atendimento simultâneo de pedidos recorrendo à interface assíncrona da API de *sockets*.
   - Desistência por timeout na recepção de pedidos (uma vez estabelecida a ligação TCP).
   - Limitação do número de ligações simultâneas que o servidor mantém com os seus clientes.
   - Suporte para *shutdown* gracioso, i.e. são rejeitados novos pedidos e aqueles cujo atendimento está em curso são servidos.
   - Funcionalidade de registo (em Logger.cs) suportada por uma thread de baixa prioridade (*logger thread*), criada para o efeito. As mensagens com os relatórios devem ser passadas das threads que servem pedidos (produtoras) para a *logger thread* usando um mecanismo de comunicação que minimize o tempo de bloqueio das *threads* produtoras. A funcionalidade de registo deve ter o mínimo de influência no tempo de serviço, admitindo-se inclusivamente a possibilidade de ignorar relatórios.

2. Implemente uma versão minimalista de cliente com interface gráfica e que realize as operações suportadas pelo servidor, apresentando os respectivos resultados.

Data limite: 15 de Fevereiro de 2012

Jorge Martins e Paulo Pereira
ISEL, 28 de Dezembro de 2011