Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Inverno de 2013/2014

Teste Final

1. [3,5] Considere a classe `UnsafeSpinCompletion`, cuja implementação é apresentada a seguir em C#:

```
public class UnsafeSpinCompletion {
      const int ALL = int.MaxValue;
      private int state = 0;

      public void Wait() {
        if (state == ALL)
          return;
        SpinWait sw = new SpinWait();
        while (state == 0)
          sw.SpinOnce();
        if (state != ALL)
          state--;
      }

      public void Complete() {
        if (state != ALL)
          state++;
      }

      public void CompleteAll() {
        state = ALL;
      }
}
```

   a) [1,5] A implementação não é *thread-safe*. Porquê?

   b) [2] Sem recorrer à utilização de primitivas de sincronização bloqueantes, apresente as alterações necessárias (em Java ou C#) para tornar a implementação *thread-safe*. Justifique as alterações realizadas.

2. [4,5] Usando monitores Java ou CLI, implemente o sincronizador *bounded bucket*, que viabiliza a recolha em lote (i.e., *bucket*) de elementos depositados individualmente. A interface pública do sincronizador é apresentada de seguida em C#.

```
public class BoundedBucket<T> {
  public BoundedBucket(int capacity);
  public bool TakeAll(int timeout, out List<T> data);
  public bool Put(T item, int timeout);
}
```

   A capacidade de cada *bucket* é especificada como parâmetro de construção (`capacity`). A operação `TakeAll` recolhe todos os elementos contidos no *bucket*, bloqueando a *thread* invocante caso o *bucket* esteja vazio. O método `Put` deposita um elemento acrescentando-o ao *bucket* actual, bloqueando a *thread* chamadora caso o *bucket* esteja cheio. Ambas as operações suportam cancelamento (i.e. interrupção) e desistência (i.e. *timeout*) das *threads* em espera.

3. [5] Usando os monitores disponíveis nas linguagens C# ou Java, implemente o sincronizador *simple I/O completion port*, que suporta uma semântica de sincronização idêntica à do sincronizador *I/O completion port* do sistema operativo *Windows*. A interface pública do sincronizador é apresentada de seguida.

```
public class SimpleIoCompletionPort<T> {
  public SimpleIoCompletionPort(int concurrencyLevel));
  public T GetQueuedCompletionStatus();
  public void PostQueuedCompletionStatus(T completionStatus);
  public void QueuedCompletionStatusDone();
}
```

O sincronizador destina-se a controlar o processamento da conclusão de operações de I/O de forma escalável, isto é, limitando o processamento simultâneo da conclusão de operações de I/O ao valor máximo especificado como parâmetro de construção (`concurrencyLevel`). Caso seja especificado zero neste parâmetro, o nível de concorrência deve ser igual ao número de processadores lógicos da máquina.

A operação `GetQueuedCompletionStatus`, que produz a informação associada à conclusão de uma operação de I/O (representada por instâncias do tipo genérico `T`), é invocada pelas *worker threads* associadas à *completion port* para obterem a informação necessária (instância de T) e assim sinalizarem o início do respectivo processamento. A sinalização da terminação do processamento associado à conclusão de uma operação de I/O é realizada através de chamadas ao método `QueuedCompletionStatusDone`. O número de operações de conclusão de I/O que num dado momento estão a ser processadas em simultâneo é portanto dado pelo número de chamadas a `GetQueuedCompletionStatus` para as quais ainda não ocorreu a chamada a `QueuedCompletionStatusDone` correspondente. A operação `GetQueuedCompletionStatus` bloqueia a as t*hreads* chamadoras caso não existam unidades de trabalho (instâncias de T) ou caso o nível máximo de concorrência tenha sido atingido.

A operação `PostQueuedCompletionStatus` é usada para a submissão de unidades de trabalho (instâncias de T). Estas unidades de trabalho, ou seja, as conclusões de I/O, devem ser processadas por ordem de chegada (FIFO), e as *worker threads* devem ser mobilizadas com critério *last-in first-out* (LIFO), para tirar melhor partido das *caches* dos processadores.

O sincronizador suporta cancelamento (i.e. interrupção) das *threads* em espera na operação `GetQueuedCompletionStatus`.

4. [4] Implemente em C# a classe `ThrottledTimeServer`, que representa um servidor concorrente de hora central (hora na máquina hospedeira do serviço). O serviço de hora central é prestado através de TCP e, na presença de pedidos de estabelecimento de ligação para o porto dado como parâmetro de construção, o servidor responde com a *string* contendo a hora actual (obtida através da expressão `DateTime.Now.ToString()`). A implementação do servidor regula o número máximo de ligações estabelecidas em simultâneo, não aceitando mais ligações caso o número limite (passado como parâmetro de construção) tenha sido atingido.

   O início da disponibilidade do servidor é marcado pela invocação ao método `Start`. Na implementação tire partido da existência das classes `TcpListener` e `TcpClient` na .NET *Framework Class Library* e tire partido das operações assíncronas disponibilizadas (baseadas no APM) para realizar atendimento concorrente de múltiplos clientes de forma eficiente. Por simplificação, não é necessário suportar o *shutdown* controlado do servidor.

5. [3] Fazendo uso da TPL, realize o método estático da classe `AsyncUtils`

```
public static Task<IEnumerable<string>> Grep(StreamReader file, string word);
```

   que produz assincronamente, e com o paralelismo possível, a sequência de linhas do ficheiro de texto (`file`) que contêm a palavra `word`.

Duração: 2 horas e 30 minutos

Carlos Martins e Paulo Pereira
ISEL, 22 de Janeiro de 2014