Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 1ª Época, Inverno de 2016/2017

---

1. [2.5] Considere a classe **`UnsafeCyclicBarrier`**, cuja implementação em *Java* se apresenta a seguir:

```
public class UnsafeCyclicBarrier {
  private final int partners;
  private int remaining, currentPhase;
  public UnsafeCyclicBarrier(int partners) {
    if (partners <= 0) throw new IllegalArgumentException();
    this.partners = this.remaining = partners;
  }
  public void signalAndAwait() {
    int phase = currentPhase;
    if (remaining == 0) throw new IllegalStateException();
    if (--remaining == 0) {
      remaining = partners; currentPhase++;
    } else {
      while (phase == currentPhase) Thread.yield();
    }
  }
}
```

   Esta implementação reflete a semântica de sincronização de uma barreira cíclica (e.g., uma barreira que pode ser usada repetidamente para sincronizar o **mesmo grupo de *threads*)**, contudo não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, uma versão *thread-safe* deste sincronizador.

2. [3,5] Implemente em C# ou *Java*, com base nos monitores implícitos ou explícitos (*Java*), o sincronizador *advertising panel*, que suporta a afixação de mensagens publicitárias pelas *threads* editoras, que ficarão expostas para consumo, durante o intervalo de tempo especificado, por parte das *threads* consumidoras. A interface pública deste sincronizador em C# é a seguinte.

```
public class AdvertisingPanel<M> where M : class {
  public void Publish(M message, int exposureTime);
  public M Consume(int timeout);    // throws ThreadInterruptedException
}
```

   A operação **`Publish`** publica uma mensagem publicitária definindo os respectivos conteúdo e tempo de exposição. O painel pode ter apenas uma mensagem afixada de cada vez, pelo que a publicação de uma mensagem pode substituir a mensagem exposta anteriormente. Sempre que é publicada uma mensagem, esta tem que ser obrigatoriamente entregue a todas as *threads* que se encontrem bloqueadas, mesmo quando o tempo de exposição da mensagem for zero (mensagens transitórias). As *threads* que pretendam consumir mensagens publicitárias invocam o método **`Consume`**, cuja execução poderá terminar: (1) devolvendo a instância do tipo **`M`** que contém uma mensagem publicitária válida; (2) devolvendo **`null`**, se expirar o intervalo de tempo especificado pelo argumento **`timeout`**, ou; (3) lançando **`ThreadInterruptedException`**, se a espera da *thread* for interrompida.

3. [4] Implemente em C# ou *Java*, com base nos monitores implícitos ou explícitos (*Java*), o sincronizador *cross exchanger*, cuja interface pública em C# é a seguinte.

```
public class CrossExchanger<T1,T2> where T1: class where T2: class {
  public T2 Exchange1(T1 mine, int timeout);    // throws ThreadInterruptedException
  public T1 Exchange2(T2 mine, int timeout);    // throws ThreadInterruptedException
}
```

   O método **`Exchange1`** é chamado pelas *threads* que pretendem oferecer uma mensagem do tipo **`T1`** (através do parâmetro **`mine`**) e receber, através do valor de retorno do método, a mensagem **`T2`** oferecida pela *thread* com

que emparelharam. O método **Exchange2** tem uma semântica semelhante, com os tipos **T1** e **T2** trocados. Note que uma *thread* que chama **Exchange1** vai emparelhar com uma *thread* que chama **Exchange2** e vice-versa. Este emparelhamento também implica que se a *thread* A receber a mensagem oferecida pela *thread* B, então a *thread* B terá obrigatoriamente que receber a mensagem oferecida pela *thread* A. A execução dos métodos **Exchange1/2** poderá terminar: (1) devolvendo a instância do tipo **T2**/**T1** que contém a mensagem obtida na troca; (2) devolvendo **null**, se expirar o intervalo de tempo especificado pelo argumento **timeout**, ou; (3) lançando **ThreadInterruptedException**, se a espera da *thread* for interrompida.

4. [7] A interface **Services** define um conjunto de operações síncronas que envolvem comunicação com um sistema externo com latência na ordem dos segundos, que usam os tipos **A**, **B**, **C**, e **D** previamente definidos. Todas estas operações não alteram estado (local ou do sistema externo), contudo a sua execução pode resultar em excepção. O método estático **Run** executa uma sequência destas operações, produzindo um resultado final ou excepção.

```
public class Execute {
  public interface Services {
    A Oper1();
    B Oper2(A a);
    C Oper3(A a);
    D Oper4(B b, C c);
  }
  public static D Run(Services svc) {
    var a = svc.Oper1();
    return svc.Oper4(svc.Oper2(a), svc.Oper3(a));
  }
}
```

   a) [3,5] A classe **APMExecute** será a variante assíncrona de **Execute** ao estilo *Asynchronous Programming Model* (APM). Implemente os métodos **BeginRun** e **EndRun** que usam a interface **APMServices** (variante APM de **Services** que não tem de apresentar). A implementação deve exprimir o paralelismo potencial que existe no método **Run**.
   NOTA: não pode usar a TPL e só se admitem esperas de controlo dentro das operações **End**, estritamente onde o APM o exige e deve considerar a ocorrência de excepções.

   b) [3,5] A classe **TAPExecute** será a variante assíncrona de **Execute**, ao estilo *Task based Asynchronous Pattern* (TAP). Usando a funcionalidade oferecida pela *Task Parallel Library* (TPL) ou pelos métodos **async** do C#, implemente o método **RunAsync**, que usa a interface **TAPServices** (variante TAP de **Services** que não tem de apresentar). A implementação deve exprimir o paralelismo potencial que existe no método **Run**.
   NOTA: na implementação não se admite a utilização de operações com bloqueios de controlo e pode ignorar a ocorrência de excepções.

5. [3] Usando os mecanismos disponíveis na *Task Parallel Library*, implemente uma versão do método estático **MapSelectedItems** que tire partido de todos os *cores* de processamento disponíveis no computador. Este método retorna o resultado da transformação (*mapping*) de todos os elementos de um enumerado que satisfaçam um dado predicado. Tenha em consideração que o processamento dos elementos do enumerado pode ser executado em paralelo, que a operação de transformação é aquela que concentra a maior componente de processamento, que a operação do método deve poder ser cancelada e que a ordem dos elementos na lista resultante não é relevante.

```
public static List<O> MapSelectedItems<I,O>(IEnumerable<I> items, Predicate<I> selector,
                                            Func<I,O> mapper, CancellationToken ctoken) {
  var result = new List<O>();
  foreach(I item in items) {
    ctoken.ThrowIfCancellationRequested();
    if (selector(item)) result.Add(mapper(item));
  }
  return result;
}
```

Duração: 2 horas e 30 minutos
ISEL, 3 de Fevereiro de 2017