Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 2ª Época, Inverno de 2016/2017

---

1. [2.5] Considere a classe **`UnsafeMessageBox`**, cuja implementação em C# se apresenta a seguir:

```
public class UnsafeMessageBox<M> where M : class {
  private class MsgHolder {
    internal readonly M msg;
    internal int lives;
  }
  private MsgHolder msgHolder = null;
  public void Publish(M m, int lvs) {msgHolder = new MsgHolder { msg = m, lives = lvs };}
  public M TryConsume() {
    if (msgHolder != null && msgHolder.lives > 0) {
      msgHolder.lives -= 1;
      return msgHolder.msg;
    }
    return null;
  }
}
```

   Esta implementação reflete a semântica de uma *message box* contendo no máximo uma mensagem que pode ser consumida múltiplas vezes, 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 *message queue*, para suportar a troca de mensagens entre *threads*. A interface pública deste sincronizador descrita em C# é a seguinte.

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

   O método **`Publish`** coloca uma mensagem na fila (**`message`**), especificando o número de vezes que a mesma pode ser consumida (**`lives`**). Podem estar simultaneamente mais do que uma mensagem na fila, devendo a respectiva entrega às *threads* consumidoras seguir o critério *first-in-first-out* (FIFO).
   As *threads* que pretendam consumir mensagens invocam o método **`Consume`**, cuja execução poderá terminar: (1) com sucesso, devolvendo a instância do tipo **`M`** que contém uma mensagem válida; (2) por *time out*, devolvendo **`null`**, se expirar o intervalo de tempo especificado pelo argumento **`timeout`**, ou; (3) por interrupção, lançando **`ThreadInterruptedException`**, se a espera da *thread* for interrompida.

3. [4] Implemente em C# ou em *Java*, com base em monitores implícitos ou explícitos (*Java*), o sincronizador *single thread pool executor*, que promove a execução sequencial dos itens de trabalho que lhe são submetidos numa única *worker thread* criada para esse efeito. A interface pública deste sincronizador, descrita em C# é a seguinte:

```
public class SingleThreadPoolExecutor {
  public object Submit(Action workItem);
  public bool Wait(object workHandle, int timeout); // throws ThreadInterruptedException
  public bool Cancel(object workHandle);
}
```

   A operação **`Submit`** submete para execução o item de trabalho passado como argumento (**`workItem`**), devolve uma instância de **`object`** que representa esse item de trabalho no contexto do executor (*work handle*) e nunca bloqueia a *thread* invocante para espera de controlo.
   A operação **`Wait`** permite às *threads* que submetem trabalho para execução sincronizarem-se com a conclusão

da execução de itens de trabalho previamente submetidos. Este método poderá terminar: (1) com sucesso, devolvendo **`true`** indicando que terminou a execução do item de trabalho especificado; (2) porque o item de trabalho foi entretanto cancelado por outra thread, devolvendo **`false`**; (3) por *time out*, devolvendo **`false`**, se expirar o intervalo de tempo especificado pelo argumento **`timeout`** sem que a execução do item de trabalho termine ou; (4) por interrupção, lançando **`ThreadInterruptedException`**, se a espera da *thread* for interrompida. A operação **`Cancel`** permite cancelar a execução do item de trabalho especificado (**`workHandle`**) no caso da respectiva execução ainda não ter sido iniciada; o método devolve **`true`** se a execução for cancelada e **`false`** no caso contrário.

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`**, **`D, I e O`**, 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(I i);
    B Oper2(I i);
    C Oper3(I i);
    D Oper4(B b, C c);
    O Oper5(A a, D d);
  }
  public static O Run(Services svc, I i) {
    return svc.Oper5(svc.Oper1(i), svc.Oper4(svc.Oper2(i), svc.Oper3(i)));
  }
}
```

   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 a implementação deverá ter em consideração 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 a implementação pode ignorar a ocorrência de excepções.

5. [3] No método **`SearchItem`**, devolve a referência para um item de dados do tipo **`T`** que satisfaça o predicado especificado com **`query`** ou **`default(T)`** se nenhum dos itens do enumerado satisfizer o predicado. O tempo máximo de pesquisa pode ser limitado usando o parâmetro **`timeout`**; quando o limite de tempo expira sem que seja encontrado nenhum item de dados que satisfaz o predicado, o método **`SearchItem`** deve lançar **`TimeoutException`**. As invocações ao predicado podem decorrer em paralelo, o que será vantajoso um vez que é nessa operação se concentra a maior parte do tempo total de execução. Tirando partido da *Task Parallel Library*, apresente uma versão de **`SearchItem`** que faça invocações paralelas ao predicado **`query`** para tirar partido dos múltiplos *cores* de processamento disponíveis. Admita que qualquer elemento para o qual **`query`** retorne **`true`** é um resultado válido, independentemente da sua posição na sequência. Assegure-se de que a execução do método termina rapidamente após ser encontrado o item a retornar ou expirar o limite de tempo de execução.

```
public static T SearchItem<T>(IEnumerable<T> items, Predicate<T> query, int timeout);
```

Duração: 2 horas e 30 minutos
ISEL, 18 de Fevereiro de 2017