Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 1ª Época, Inverno de 2018/2019

---

1. [2,5] Considere a definição em C# da classe **`UnsafeSpinCompletion`**, como uma tentativa para implementar o sincronizador *completion* implementado no *kernel* do sistema operativo *Linux*.

```
public class UnsafeSpinCompletion {
  private const int OPEN = -1;
  private int state = 0;

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

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

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

   Esta classe ilustra um algoritmo que implementa o sincronizador *completion*, contudo não é *thread safe*. Sem usar *locks*, implemente, em C# ou *Java* , uma versão *thread safe* desta classe.

2. [4] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *keyed channel*, para suportar a comunicação entre *threads* através de mensagens que têm associada uma chave. A interface pública deste sincronizador, em C#, é a seguinte:

```
public class KeyedChannel<K, T> where T: class {
  public void Put(T msg, K key);
  public T Take(K key, int timeout);
}
```

   O método **`Put`** entrega uma mensagem ao canal, associando-lhe a respectiva chave. Invocando o método **`Take`**, a respectiva *thread* manifesta a intenção de receber a próxima mensagem cuja chave seja igual à especificada, ficando bloqueada se não existir nenhuma mensagem disponível. O método **`Take`** termina: (a) retornando a mensagem, quando esta for recebida; (b) retornando **`null`** se for excedido o limite especificado para o tempo de espera, e; (c) lançando **`ThreadInterruptedException`** quando a espera da *thread* é Interrompida. Tanto na produção como no consumo das mensagens deve ser respeitada a disciplina FIFO. Além disso, a implementação do sincronizador deve optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

3. [4] Implemente em *Java* ou C#, como base nos monitores implícitos ou explícitos, o sincronizador *synchronous queue with replication*, cuja interface publica, em C#, é a seguinte:

```
public class SynchronousQueueWithReplication<T> {
  public SynchronousQueueWithReplication(int nOfReplicas);
  public bool Send(T sentMsg, int timeout);
  public bool Receive(int timeout, out T recvdMsg);
}
```

Este sincronizador permite a comunicação entre *threads* produtoras e *threads* consumidoras com semântica síncrona, garantindo a entrega de cada mensagem a um número predefinido de consumidores. A operação **`Send`** entrega uma mensagem à fila (**`sentMsg`**), e termina: (a) devolvendo **`true`** se a mensagem for entregue a **`nOfReplicas`** *thread* consumidoras; (b) devolvendo **`false`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`ThreadInterruptedException`**, se o bloqueio da *thread* for interrompido. Quando a operação **`Send`** falha por *timeout* ou interrupção, a respectiva mensagem deve ser descartada. A operação **`Receive`** permite receber uma mensagem da fila, e termina: (a) devolvendo **`true`** se receber uma mensagem, sendo esta devolvida através do parâmetro **`recvdMsg`**; (b) devolvendo **`false`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`ThreadInterruptedException`**, se o bloqueio da *thread* for interrompido. Uma mensagem enviada para a fila é entregue a exactamente **`nOfReplicas`** *threads* consumidoras ou então não é entregue a nenhuma *thread* consumidora.

4. [6] Considere o seguinte método:

```
public R MapReduce(T[] elems, R initial)  {
  for (int i = 0 ; i < elems.Length ; ++i)
    initial = Reduce(Map(elems[i]), initial);
  return initial;
}
```

   a. [2] Tendo em consideração as características das aplicações e dos actuais sistemas computacionais, diga sucintamente quais são as principais vantages da utilização da programação assíncrona.

   b. [4] O método **`Map`** é uma função sem efeitos colaterais e passível de múltiplas execuções em paralelo. O método **`Reduce`** é associativo e comutativo, podendo ser executado por qualquer ordem, contudo não pode existir mais do que uma execução em simultâneo. Realize uma versão assíncrona do método **`MapReduce`** seguindo o padrão TAP (*Task-based Asynchronous Pattern*) usando a TPL e/ou os métodos assíncronos do C#. Assuma que tem disponível versões TAP dos métodos **`Map`** e **`Reduce`**. Tire partido do paralelismo potencial existente, valorizando-se soluções que iniciem as operações **`Reduce`** o mais cedo possível.

5. [3,5] De modo a realizar análise estatística de texto foi definido o método **`TextAnalyzer`**, cujo código é apresentado abaixo na versão sequencial. As chamadas à função de conversão (**`conv`**) são independentes entre si pelo que podem decorrer em paralelo; a operação de agregação de resultados (**`agg`**) é comutativa e associativa sendo o seu elemento neutro obtido com a função **`initial`**; a função **`goal`** determina se objectivo da análise já foi atingido. Tirando partido da *Task Parallel Library*, apresente uma versão deste método, **`ParallelTextAnalyzer`**, que tire partido de todos os *cores* de processamento disponíveis, e que termine o mais rapidamente possível após determinar que o objectivo foi atingido.

```
public static R TextAnalyzer<R>(IEnumerable<String> text, Func<String,R> conv,
                                 Func<R,R,R> agg, Func<R,bool> goal,Func<R> initial)  {
  R result = initial();
  foreach (var str in text) {
    result = agg(result, conv(str));
    if (goal(result))
      break;
  }
  return result;
}
```

Duração: 2 horas e 30 minutos
ISEL, 4 de janeiro de 2019