Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 1ª Época, Inverno de 2017/2018

---

1. [2,5] Considere a definição em C# da classe `UnsafeCLHSpinLock`, como uma tentativa para implementar um *CLH queued spinlock* (CLH deriva do nome dos seus autores: *Craig*, *Landin* e *Hagersten*).

```
public class UnsafeCLHSpinLock {
  public class CLHNode {
    public bool succMustWait = true;  // The default is to wait for a lock
  }
  private CLHNode tail;       // the tail of the wait queue; when null the lock is free
  public CLHNode Acquire() {
    CLHNode myNode = new CLHNode();
    // insert my node at tail of queue and get my predecessor
    CLHNode predecessor = tail; tail = myNode;
    // If there is a predecessor spin until the lock is free; otherwise we got the lock
    if (predecessor != null) {
      SpinWait sw = new SpinWait();
      while (predecessor.succMustWait) sw.SpinOnce();
    }
    return myNode;
  }
  public void Release(CLHNode myNode /* the node returned from corresponding Acquire */) {
    // If we are the last node on the queue, then set tail to null; else release successor
    if (tail == myNode)
      tail = null;
    else
      myNode.succMustWait = false ;  // Release our successor
  }
}
```

   Esta classe ilustra o algoritmo do *CLH spinlock*, 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 *batch builder*, cuja interface pública em *Java* é a seguinte:

```
public class BatchBuilder {
    public BatchBuilder(int batchSize);
    public List<Object> await(Object value, int timeout) throws InterruptedException;
}
```

   A operação **`await`** fornece um valor do tipo **`Object`** e espera até que estejam presentes no sincronizador um grupo de **`batchSize`** *threads*, incluindo a própria, e termina: (a) devolvendo uma lista com os valores fornecidos por todas as *threads* do grupo; (b) devolvendo **`null`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido. A título de exemplo, considerando **`batchSize`** igual a 3, se a *thread* T1 retorna da operação com os valores fornecidos pelas *threads* T1, T2 e T3, então as *threads* T2 e T3 tem de sair da operação com exactamente a mesma lista de valores. Caso a operação termine com desistência ou excepção, o valor fornecido pela respectiva *thread* não pode ser usado em nenhum agrupamento.

3. [3,5] Implemente, em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *enhanced transient event*, cuja interface pública em *Java* é a seguinte:

```
public class EnhancedTransientEvent {
  public boolean await(int timeout) throws InterruptedException;
  public void signal(int maxWakeups);
}
```

A operação **`await`** bloqueia a *thread* invocante a aguardar uma sinalização, e termina: (a) devolvendo **`true`** se a *thread* for desbloqueada pela operação **`signal`**; (b) devolvendo **`false`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido. A operação **`signal`** desbloqueia no máximo **`maxWakeups`** *threads* bloqueadas pela operação **`await`**; a operação **`signal`** não afecta operações **`await`** futuras. A ordem de libertação deve respeitar o critério FIFO (*first-in-first-out*) e a implementação deve minimizar o número de comutações de contexto.

4. [7] Considere o seguinte método:

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

O método **`Map`** é uma função sem efeitos colaterais e passível de múltiplas execuções em paralelo. O método **`Reduce`** não é associativo e tem que ser executado em série e pela ordem definida.

   a. [3,5] Realize uma versão assíncrona do método **`MapReduce`** usando o padrão APM (*Asynchronous Programming Model*), assumindo que tem disponível versões APM dos métodos **`Map`** e **`Reduce`**. Tire partido do paralelismo potencial existente, realizando as operações **`Map`** em paralelo e, após todas estarem concluídas, realizando, em série, assincronamente as operações **`Reduce`**.

      Nota: não pode usar a TPL e só são admitidas esperas de controlo dentro das operações **`EndXxx`**, estritamente onde o APM o exige.

   b. [3,5] 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 a sequência de operações **`Reduce`** antes de estarem concluídas todas as operações **`Map`**.

5. [3] Usando os mecanismos disponíveis na *Task Parallel Library*, implemente o seguinte método:

```
public T[] GetTopN<T>(IEnumerable<T> items, int topSize) where T : IComparable<T>;
```

Este método retorna um *array*, com dimensão máxima **`topSize`**, contendo os maiores elementos do enumerável **`items`** e deve utilizar todos os *cores* de processamento disponíveis no computador. Na sua implementação, considere que tem disponível a classe **`BoundedOrderedQueue<T>`**, não *thread safe*, que mantém uma fila ordenada de elementos com a capacidade máxima especificada na construção e cuja especificação é a seguinte:

```
public class BoundedOrderedQueue<T>  {
  public BoundedOrderedQueue(int capacity);
  public void Add(T item);
  public void Merge(BoundedOrderedQueue<T> other);
  public T[] ToArray();
}
```

Duração: 2 horas e 30 minutos
ISEL, 6 de Janeiro de 2018