Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

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

---

1. [2,5] Considere a definição em C# da classe **`UnsafeSpinWindowsMutex`**, como uma tentativa para implementar o sincronizador *mutex* adoptando o algoritmo usado no sistema operativo *Windows*.

```
public class UnsafeSpinWindowsMutex {
  private int signalState = 1; // 1 when the mutex is free, <= 0 when the mutex is busy
  private Thread owner;        // null or the mutex's owner thread

  public void Acquire() {
    SpinWait sw = new SpinWait();
    do {
      if (signalState > 0 || owner == Thread.CurrentThread) {
        if (signalState == Int32.MinValue) throw new MutexLimitExceededException();
        if (--signalState == 0)
           owner = Thread.CurrentThread;
        return;
      }
      sw.SpinOnce();
    } while (true);
  }

  public void Release() {
    if (owner != Thread.CurrentThread) throw new MutexNotOwnedException();
    if (signalState == 0)  // if the mutex becomes free, first, reset the owner
      owner = null;
    signalState += 1;
  }
}
```

   Esta classe ilustra o algoritmo que implementa o *mutex*, 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 *synchronous message queue,* cuja interface pública em C# é a seguinte:

```
public class SynchronousMessageQueue<T> {
  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. A operação **`Send`** entrega uma mensagem à fila (**`sentMsg`**), e termina: (a) devolvendo **`true`** se a mensagem for recebida por uma *thread* consumidora; (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.

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

```
public class KeyedExchanger<T> {
   public bool Exchange(int pairkey, T mydata, int timeout, out T yourData);
}
```

Este sincronizador suporta a troca de informação entre pares de *threads* identificados por uma chave. As *threads* que utilizam este sincronizador manifestam a sua disponibilidade para iniciar uma troca invocando o método **`Exchange`**, especificando a identificação do par (**`pairkey`**), o objecto que pretendem entregar à *thread* parceira (**`myData`**) e, opcionalmente, o tempo limite da espera pela troca (**`timeout`**). O método **`Exchange`** termina: devolvendo **`true`**, quando é realizada a troca com outra *thread*, sendo o objecto por ela oferecido retornado através do parâmetro **`yourData`**; devolvendo **`false`**, se expirar o limite do tempo de espera especificado, ou; lançando **`ThreadInterruptedException`** quando a espera da *thread* for interrompida.

Nota: Se implementar o sincronizador em *Java* altere adequadamente a assinatura do método `Exchange`.

4. [6] A interface **`IServices`** define os serviços síncronos disponibilizados por uma empresa que oferece um serviço de estudos de opinião, capaz de responder a perguntas de forma referendária (sim/não). O método **`GetVoters`** devole o conjunto dos servidores que, no momento, têm opinião sobre a pergunta especificada. O método **`GetAnswer`** permite obter a resposta do servidor especificado. O método estático **`Query`** devolve a resposta à pergunta especificada e termina assim que obtiver uma maioria de respostas positivas ou negativas.

```
public class Question4 {
  public interface IServices {
    Uri[] GetVoters(String question);
    bool GetAnswer(Uri voter, String question);
  }

  public static bool Query(IServices svc, String question) {
    Uri[] voters = svc.GetVoters(question);
    int agree, n;
    for (agree = 0, n = 1; n <= voters.Length; n++) {
      agree += svc.GetAnswer(voters[n - 1], question) ? 1 : 0;
      if (agree > (voters.Length / 2) || n - agree > (voters.Length / 2))
        break;
    }
    return agree > voters.Length / 2;
  }
}
```

   a. [2] Descreva sucintamente quais são as principais diferenças entre a invocação síncrona e a invocação assíncrona de operações. Explicite quais são as implicações que essas diferenças têm na utilização dos recursos computacionais quando se executam em simultâneo múltiplas operações I/O *bound*.

   b. [4] Usando a TPL e/ou os métodos assíncronos do C#, implemente o método **`QueryAsync`**, versão assíncrona do método **`Query`**, seguindo o padrão TAP (*Task-based Asynchronous Pattern*). Assuma que tem disponível a versão TAP da interface **`IServices`** e que o método **`GetAnswerAsync`** aceita um terceiro argumento, do tipo **`CancellationToken`**, que permite solicitar o cancelamento a operação assíncrona.

5. [3,5] O método **`Select`**, apresentado a seguir, selecciona e devolve pelo menos **`count`** itens de dados da colecção **`items`** que pertençam também à colecção **`keys`**. Tirando partido da *Task Parallel Library*, implemente o método **`ParallelSelect`** para realizar o mesmo processamento mas de modo a tirar partido de todos os *cores* de processamento disponíveis.

```
static List<T> Select<T>(IEnumerable<T> items, IEnumerable<T> keys, int count,
                         CancellationToken ct) {
  List<T> result = new List<T>();
  foreach (T item in items) {
    ct.ThrowIfCancellationRequested();
    foreach(T key in keys)
      if (item.Equals(key)) { result.Add(item); break; }
    if (result.Count >= count)
      break;
  }
  return result;
}
```

Duração: 2 horas e 30 minutos