Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente
Teste Global da Época de Recurso, Verão de 2018/2019

---

1. [3] Considere a definição em *Java* da classe **`UnsafeSpinSingleConsumerQueue`**, como uma tentativa para implementar um mecanismo de comunicação entre *threads* em cenários com várias *threads* produtoras e apenas uma *thread* consumidora (semântica detalha no enunciado do exercício 2).

```
public class UnsafeSpinSingleConsumerQueue {
  private class Node {
    internal Node next;
    internal readonly T data;
    internal Node(T d) { data = d; }
  }
  private Node productions = null;
  public void Put(T data) {
    Node node = new Node(data);
    node.next = productions;
    productions = node;
  }
  public List<T> Take(int timeout) {
    TimeoutHolder th = new TimeoutHolder(timeout);
    while (productions == null && th.Value > 0) Thread.Yield();
    if (th.Value <= 0) return null;
    Node prods = productions;
    productions = null;
    List<T> toRet = new List<T>();
    do {
      toRet.Add(prods.data);
    } while ((prods = prods.next) != null);
    return toRet;
  }
}
```

   Esta classe ilustra um algoritmo que implementa o sincronizador *single consumer queue*, contudo não é *thread safe*. Sem usar *locks*, implemente, em C# ou *Java*, uma versão *thread safe* desta classe (**`SafeXxx`**).

2. [4] Implemente, em *Java* ou C#, como base nos monitores implícitos ou explícitos, o sincronizador *single consumer queue* que se destina a suportar a comunicação entre *threads* em cenários com múltiplas *threads* produtoras e apenas uma *thread* consumidora e cuja interface pública em C# é a seguinte:

```
public class SingleConsumerQueue<T> {
  public void Put(T data);
  public List<T> Take(int timeout);
}
```

   A operação **`Put`** acrescenta um item de dados à fila e nunca bloqueia a *thread* invocante. A operação **`Take`** bloqueia a *thread* invocante quando a fila está vazia, e termina: (a) com sucesso, retornando uma lista com todos os itens de dados que se encontram na fila; (b) devolvendo **`null`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`ThreadInterruptedException`**, se o bloqueio da *thread* for interrompido. (Considere que a ordem dos itens de dados na lista devolvida pelo método **`Take`** podem não respeitar a ordem com que os mesmos foram entregues à fila.)

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

```
public class NotificationEvent {
  public NotificationEvent(boolean signaled);
  public boolean await(long timeout) throws InterruptedException;
```

```
  public void set();
  public void pulse();
  public void reset();
}
```

O evento pode estar num de dois estados: sinalizado ou não sinalizado. A operação **`await`** sincroniza a *thread* invocante com a sinalização do evento, e termina: (a) devolvendo **`true`**, se o evento tiver sido sinalizado e não modifica o estado do evento; (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 **`set`** sinalza o evento libertando todas as *threads* nele bloqueadas e deixando o evento no estado sinalizado. A operação **`pulse`** liberta todas as *threads* que se encontrem bloqueadas no evento, deixando-o no estado não sinalizado. A operação **`reset`** coloca o evento no estado não sinalizado.

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 Polling {
  public interface IServices {
    Url[] GetVoters(String question);
    bool GetAnswer(Url voter, String question, CancellationToken ctoken);
  }
  public static bool Query(IServices svc, String question) {
    CancellationTokenSource cts = new CancellationTokenSource();
    Url[] voters = svc.GetVoters(question);
    int agree, n;
    for (agree = 0, n = 1; n <= voters.Length; n++) {
      agree += svc.GetAnswer(voters[n - 1], question, cts.Token) ? 1 : 0;
      if (agree > (voters.Length / 2) || n - agree > (voters.Length / 2)) break;
    }
    if (n < voters.Length) cts.Cancel();
    return agree > voters.Length / 2;
  }
}
```

   a. [2] Enuncie quais os critérios usados pelo *thread pool* do *.NET Framework* para injectar novas *worker threads* e para deixar terminar *worker threads* activas. Estes critérios visam optimizar a utilização de recursos do sistema, quais?

   b. [4] Usando a TPL e/ou os métodos assíncronos do C#, implemente o método **`QueryAsync`**, versão assíncrona do mé todo **`Query`**, seguindo o padrão TAP (*Task-based Asynchronous Pattern*). Assuma que tem disponível a versão TAP da interface **`IServices`**.

5. [3] Usando os mecanismos disponíveis na *Task Parallel Library* para suporte à programação paralela, implemente o método **`SelectUntilCountParallel`** de forma a utilizar todos os *cores* de processamento disponíveis. Este método seleciona até **`count`** elementos arbitrários do enumerável **`items`** que obedeçam ao predicado especificado por **`selector`**; se os elementos que satisfazem o predicado for inferior a **`count`**, o método devolve todos esses elementos. Considere que as chamadas a **`selector`** podem ser feitas em paralelo, que a operação deve ser cancelável através do parâmetro **`ctoken`** e que o método deve retornar o mais rapidamente possível assim que forem selecionados os **`count`** elementos.

```
static List<T> SelectUntilCount<T>(IEnumerable<T> items, Predicate<T> selector,
                                   int count, CancellationToken ctoken);
```

Duração: 2 horas e 30 minutos
ISEL, 10 de Julho de 2019