Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 1ª Época,Verão de 2014/2015

---

1. [3] Considere a classe `UnsafeTicketSpinLock`, como uma tentativa para implementar um *ticket spin lock* (usado pelo *kernel* do *Linux*).

```
public class UnsafeTicketSpinLock {
  private int ticket;
  private int users;

  public void Lock() {
    SpinWait sw = new SpinWait();
    int me = users++;
    while (me != ticket)
      sw.SpinOnce();
  }

  public void Unlock() { ticket++; }
}
```

   Esta classe ilustra o algoritmo do *ticket spin lock*, mas não é *thread-safe*. Sem recorrer à utilização de *locks*, implemente a classe *thread-safe* `TicketSpinLock`, com a mesma semântica de sincronização e introduzindo as mesmas barreiras de memória que os *locks* introduzem nos vários modelos de memória (e.g., JMM).

2. [5] Implemente em Java ou C#, usando monitores implícitos ou explícitos (Java), o sincronizador *transfer queue* que suporta a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico `E`. A interface pública do sincronizador descreve-se a seguir em C#.

```
public class TransferQueue<E> where E : class {
   public void Put(E msg);      // adiciona uma mensagem à fila
   public void Transfer(E msg); // adiciona uma mensagem à fila e aguarda que
                                // a mesma seja recebida

   public E Take();             // obtém uma mensagem da fila
}
```

   O método `Put` entrega uma mensagem à fila, nunca bloqueando por se considerar que a fila tem uma capacidade ilimitada. O método `Transfer` entrega uma mensagem à fila e aguarda que a mesma seja recebida por uma das *threads* consumidoras. As mensagens devem ser entregues às *threads* consumidoras por ordem de chegada (FIFO), sendo irrelevante a ordem pela qual as *threads* servidoras são seleccionadas. O sincronizador deve suportar a interrupção das *threads* bloqueadas no método `Transfer`.

3. [5] Implemente em Java ou C# utilizando os monitores implícitos ou os monitores explíctos (Java), o sincronizador `NtKeyedEvent` (sincronizador utilizado no *kernel* do *Windows NT*), cujo objectivo é suportar a sincronização de pares de *threads* que são identificadas através de uma chave. A seguir mostramos a interface pública do sincronizador, definida em C#.

```
public class NtKeyedEvent {
   public bool Release(object key, int timeout);
   public bool Wait(object key, int timeout);
}
```

   O sincronizador suporta duas operações, ambas bloqueantes. A operação `Release` bloqueia a *thread* invocante até que seja chamada a operação `Wait` com a mesma chave. A operação `Wait` bloqueia a *thread* invocante até que seja chamada a operação `Release` com a `mesma` chave. Ambas as operações podem terminar: com sucesso, quando um par de *threads* invoca `Release` e `Wait` com a mesma `chave`; com insucesso, quando

qualuqer das *threads* blouqeadas por `Wait` ou `Release` é interrompida, ou tenha sido atingido o tempo máximo de espera especificado. (Considera-se que as chaves utilizadas simultaneamente pelos pares de *threads* em sincronização nunca são iguais.)

4. [7] A interface `Services` representa os serviços síncronos disponibilizados por um sistema de gestão de servidores, geridos por um *provider*, suportando todos o mesmo tipo de serviços. O método síncrono `GetFastServer` usa as operações de `Services` para determinar qual o servidor que apresenta um tempo de serviço médio menor (definido através de uma instância do tipo `Tuple<Uri, int>`, onde consta o `Uri` do servidor e o tempo médio de serviço). Todas as operações de `Services` envolvem comunicações com servidores, através de redes de dados, pelo que seria vantajosa a disponibilização da variante assíncrona do método `GetFasterServer`.

```
public class SyncOps {
   public interface Services {
     Uri[] GetAvailableServers(Uri provider);
     int GetAverageServiceTime(Uri server);
   }

   public static Tuple<Uri, int> GetFasterServer(Services svc, Uri provider) {
      Uri[] servers = svc.GetAvailableServers(provider);
      Tuple<Uri, int> result = null;
      int fastServiceTime = int.MaxValue;
      for (int i = 0; i < servers.Length; i++) {
        int ast = svc.GetAverageServiceTime(servers[i]);
        if (ast < fastServiceTime) {
          fastServiceTime = ast;
          result = new Tuple<Uri, int>(servers[i], ast);
      }
      return result;
   }
}
```

a) [4] A classe `APMOps` será a variante assíncrona de `SyncOps`, seguindo o estilo *Asynchronous Programming Model* (APM). Apresente os métodos `BeginGetFasterServer` e `EndGetFasterServer`, que usam a interface `APMServices` (variante APM de `Services` que não tem de apresentar).
NOTA: não pode usar a TPL e só se admitem esperas de controlo dentro das operações `End`, estritamente onde o APM o exige.

b) [3] A classe `TAPOps` será a variante assíncrona de `SyncOps`, seguindo o estilo *Task-based Asynchronous Pattern* (TAP). Tirando partido da *Task Parallel Library* (TPL) ou dos métodos `async` do C#, implemente o método `GetFasterServerAsync`, que usa a interface `TAPServices` (variante TAP de `Services` que não tem de apresentar).
NOTA: na implementação não se admite a utilização de operações com bloqueios de controlo.

Duração: 2 horas e 30 minutos
ISEL, 1 de Julho de 2015