Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
**Programação Concorrente**
Teste Global de 2ª Época, Verão de 2016/2017

---

1. [2.5] Considere a classe **`UnsafeLock`**, cuja implementação em C# se apresenta a seguir:

```
public class UnsafeLock {
  private int state = -1;            // state is -1 when the Lock is free
  private EventWaitHandle waitEvent; // event where waiters are blocked - lazy created
  private EventWaitHandle Event {
    get {
      if (waitEvent == null) waitEvent = new AutoResetEvent(false);
      return waitEvent;
    }
  }
  public void Enter() {
    if (++state == 0) return; // lock was free, we acquired it
    Event.WaitOne();  // lock is busy, block the current thread
  }
  public void Leave() {
    if (--state >= 0)
      Event.Set();   // at least one thread is waiting, grant ownership to one of them
  }
}
```

   Esta classe implementa um *lock* de posse exclusiva em ambiente *Windows*, contudo não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, uma versão *thread-safe* deste sincronizador.

2. [4] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *transient signal* que suporta o envio de notifcações entre *threads* e cuja interface pública em *Java* é a seguinte:

```
public class TransientSignal {
  public int await(long timeout) throws InterruptedException;
  public void signal(int reason);
  public void signalAll(int reason);
}
```

   O método **`await`** bloqueia sempre a *thread* invocante e termina quando: (a) ocorrer uma notificação (com **`signal`** ou **`signalAll`**), devolvendo o valor do argumento **`reason`** passado ao método de notificação; (b) expirar o limite de tempo de espera especificado, devolvendo **`-1`**, ou; (c) for interrompido o bloqueio da *thread*, lançando **`InterruptedException`**. O método **`signal`** desbloqueia a *thread* que se encontrar bloqueada há mais tempo (FIFO). O método **`signalAll`** desbloqueia todas as *threads* que se encontrem bloqueadas pelo método **`await`**. Os métodos **`signal`** e **`signalAll`** especificam a razão da notificação com o argumento **`reason`**. Quando não existirem *threads* bloqueadas, as notificações não terão qualquer efeito.

3. [4] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *transfer queue* que suporta a comunicação entre *threads* produtoras e consumidoras e cuja interface pública em C# é a seguinte:

```
public class TransferQueue<T> {
  public void Put(T msg);
  public bool Transfer(T msg, int timeout);
  public bool Take(int timeout, out T rmsg);
}
```

   O método **`Put`** entrega uma mensagem à fila, e nunca bloqueia a *thread* invocante. O método **`Transfer`** entrega uma mensagem à fila e aguarda que a mesma seja recebida por outra *thread* (via método **`Take)`** e termina: (a) devolvendo **`true`**, se a mensagem for recebida por outra *thread*; (b) devolvendo **`false`**, se expirar o limite do tempo de espera especificado sem que a mensagem seja recebida por outra *thread*, ou; (c) lançando **`ThreadInterruptedException`** quando a espera da *thread* for interrompida. (Quando o método **`Transfer`** retorna sem sucesso, a respectiva mensagem não deve ficar na fila.) O método **`Take`** permite receber uma

mensagem da fila e termina: (a) devolvendo **true** se for recebida uma mensagem, sendo esta devolvida através do parâmetro **rmsg**; (b) devolvendo **false**, se expirar o limite do tempo de espera especificado, ou; (c) lançando **ThreadInterruptedException** quando a espera da *thread* é interrompida. Na implementação do sincronizador deve procurar minimizar o número de comutações de *threads* em todas as circunstâncias, usando as técnicas estudadas na unidade curricular.

4. [7] A interface **IServices** define o conjunto de serviços síncronos disponibilizados por uma empresa que oferece aos seus clientes a possibilidade de executar processamento nos seus servidores. O método **Login** estabelece uma sessão de utilização em proveito do utilizador especifcado por **uid**, lançando a excepção **IllegalUserException** se for especificado um utilizador inválido. O método **ExecService** executa o serviço especificado no âmbito da sessão especificada e pode lançar a excepção **ServerTooBusyException**. Por fim, o método **Logout** encerra a sessão iniciada com o método **Login**. O método estático **ExecServices** executa, sincronamente, um conjunto de serviços e pode terminar: (a) lançando a excepção **IllegalUserException** se não for possível iniciar a sessão, ou; (b) devolvendo o *array* com as respostas à execução dos serviços (devolvendo **null** como resposta aos serviços cuja execução lançou excepção). O método **ExecServices** só termina após terminado o **Logout**; contudo, devem ser ignoradas as excepções lançadas por este método. Ainda que o método **ExecServices** só termine após executar o método **Logout**, deverão ser ignoradas as excepções lançadas por este método.

```
public class Exec {
   public interface IServices {
     Session Login(UserID uid);
     Response ExecService(Session session, Request request);
     void Logout(Session session);
   }
   public static Response[] ExecServices(IServices svc, UserID uid, Request[] requests) {
     Session session = svc.Login(uid);
     Response[] responses = new Response[requests.Length];
     for (int i = 0; i < requests.Length; i++)
       try { responses[i] = svc.ExecService(session, requests[i]); } catch {};
     svc.Logout(session);
     return responses;
   }
 }
```

   a. [3,5] A classe **APMExec** será a variante assíncrona de **Exec** ao estilo *Asynchronous Programming Model* (APM). Implemente os métodos **BeginExecServices** e **EndExecServices** que usam a interface **IAPMServices** (variante APM de **IServices** que não tem de apresentar).

   b. [3,5] A classe **TAPExec** será a variante assíncrona de **Exec**, ao estilo *Task based Asynchronous Pattern* (TAP). Usando a funcionalidade oferecida pela *Task Parallel Library* (TPL) ou pelos métodos **async** do C#, implemente o método **ExecServicesAsync**, que usa a interface **ITAPServices** (variante TAP de **IServices** que não tem de apresentar).

5. [2,5] O método **SelectIetms**, apresentado a seguir, selecciona e devolve os itens de dados da colecção **items** que pertencem à colecção **keys**. Tirando partido da *Task Parallel Library*, implemente o método **ParallelSelectIetms** 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, 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; }
  }
  return result;
}
```

Duração: 2 horas e 30 minutos
ISEL, 18 de Julho de 2017