Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 2ª Época, Inverno de 2015/2016

---

1. [3] Considere a classe `UnsafeSpinWindowsSemaphore` cuja implementação em *Java* é apresentada a seguir:

```
class UnsafeSpinWindowsSemaphore {
  private int limit, count;

  public UnsafeSpinWindowsSemaphore(int count, int limit) {
    if (count < 0 || count > limit) throw new IllegalArgumentException("bad count/limit");
    this.count = count; this.limit = limit;
  }

  public void await() {
    while (count == 0) Thread.yield();
    --count;
  }

  public void release(int rcount) {
    if (count + rcount < count || count + rcount > limit)
      throw new IllegalArgumentException("limit exceeded");
    count += rcount;
  }
}
```

   Esta implementação reflete a semântica de sincronização do semáforo no sistema operativo *Windows*, mas não é *thread-safe*. Implemente em *Java*, sem utilizar *locks*, uma versão *thread-safe* deste semáforo.

2. [3] Implemente em C#, com base nos respectivos monitores implícitos, o sincronizador *create once lock*, cuja interface pública se apresenta a seguir:

```
class CreateOnceLock {
   public boolean TryCreate();
   public void OnCreationSucceeded();
   public void OnCreationFailed();
}
```

   Este tipo de *lock* é usado para promover *one-time-creation* de instâncias *singleton* de qualquer tipo de dados com código semelhante ao que se apresenta a seguir:

```
public static T LazyInit<T>(ref T target, Funct<T> factory, CreateOnceLock coLock) {
  if (coLock.TryCreate())
    try { target = factory(); coLock.OnCreateSucceded(); }
    catch { coLock.OnCreationFailed(); throw; }
  return target;
}
```

   A operação `TryCreate` é potencialmente bloqueante e pode concluir: (1) devolvendo `true`, se a responsabilidade da criação do objecto for atribuída à *thread* invocante; (2) devolvendo `false` se a criação do objecto já foi concluída por outra *thread*, ou; (3) lançando `ThreadInterruptedException` quando o bloqueio da *thread* é interrompido. A invocação da operação `OnCreationSucceeded` sinaliza que o objecto foi criado com sucesso e marca o instante a partir do qual todas as chamadas anteriores e futuras ao método `TryCreate` devolvem `false`. O método `OnCreationFailed` sinaliza que a criação em curso falhou, levando o sincronizador ao seu estado inicial, isto é, o sincronizador deverá atribuir a responsabilidade da criação a outra das *threads* que invocou ou invocará o método `TryCreate`. A implementação do sincronizador deve processar correctamente a interrupção das *threads* nele bloqueadas e optimizar o número de comutações de *thread* que ocorrem nas várias circunstâncias.

3. [4] Implemente em *Java*, com base nos monitores implícitos ou explícitos, o sincronizador *high priority sequential executor*, que executa as funções que lhe são submetidas numa única *thread* dedicada de alta prioridade que o

sincronizador deve criar para o efeito, assegurando uma execução sequencial dos itens de trabalho que lhe são submetidos. A interface pública deste sincronizador, em *Java*, é a seguinte:

```
class HighPrioritySequentialExecutor<T> {
   public T execute(Callable<T> toCall) throws Exception, InterruptedException;
   public void shutdown();
}
// as defined in the java.util.concurrent package
interface Callable<V> { public V call() throws Exception; }
```

As *threads* que pretendem executar funções usando o sincronizador invocam o método `execute` especificando a função a executar através do parâmetro `toCall`. Este método bloqueia sempre a *thread* invocante até que a função especificada seja executada pela *thread* dedicada e pode terminar: (1) normalmente, devolvendo a instância do tipo `T` devolvida pela função, ou; (2) excepcionalmente, lançando a mesma excepção que foi lançada aquando da chamada à função. Até ao momento em que a *thread* dedicada considerar uma função para execução, é possível interromper a execução do método `execute`; contudo, se a interrupção ocorrer depois da função ser aceite para execução, o método `execute` deve ser processado normalmente, sendo a interrupção memorizada de forma a que possa vir a ser lançada pela *thread* mais tarde. A chamada ao métofo `shutdown` coloca o executor em modo *shutdown*. Neste modo, todas as chamadas ao método `execute` lançarão a excepção `IllegalStateException`; contudo, todas as submissões para execução feitas antes do *shutdown* devem ser processadas normalmente. O método `shutdown` deverá bloquear a *thread* invocante até que sejam executados todos os itens de trabalho submetidos antes do *shutdown*.

4. [7] A interface `Service` (não apresentada) disponibiliza serviços base de um sistema de gestão de equipamentos, em versões síncronas e assíncronas, quer no modelo *Asynchronous Programming Model* (APM), quer no modelo *Task-based Asynchronous Pattern* (TAP). A classe `Control` disponibiliza a operação composta `CheckDeviceAsync`, também assíncrona, que invoca vários operações de `Service` internamente. Verifica-se que o uso desta operação nos servidores do sistema resulta num consumo considerável de recursos, principalmente nos momentos em que há mais pedidos em curso simultaneamente. Para além disso, observam-se ocasionalmente resultados incorrectos.

```
public class Control {
   public static Task<DevInfo> CheckDeviceAsync(Service svc, int devId) {
      DevState lastKnownState = null;
      Task.Run(() => { lastKnownState = svc.GetKnownDevState(devId); });
      return Task.Run(() => {
         DevAddr addr = svc.GetDevAddr(devId);
         DevReport devReport = svc.GetCurrDevReport(addr);
         return new DevInfo(lastKnownState, devReport); });
   }
}
```

   a. [1.5] Sabendo que as operações de `Service` consistem essencialmente em I/O e analisando a implementação fornecida, qual é o defeito grave de implementação de `CheckDeviceAsync` que leva a um elevado consumo de recursos quando é intensivamente utilizado em vários pedidos em simultâneo e qual pode ser a causa dos resultados incorrectos ocasionais? Justifique devidamente as duas respostas.
   b. [2] Apresente uma implementação corrigida de `CheckDeviceAsync`, utilizando devidamente a *Task Parallel Library* (TPL) e/ou os métodos *async* do C#, que tire partido das operações assíncronas de `Service` e produza resultados correctos em todas as execuções.
   c. [3.5] Acrescente à classe Control as operações `BeginCheckDevice` e `EndCheckDevice`, para que a operação `CheckDevice` também possa ser utilizada de acordo com o modelo APM.
      NOTA: não pode usar a TPL e só se admitem esperas de controlo dentro da operação `End`, estritamente onde o APM o exige.

5. [3] No método `FindItem`, apresentado a seguir, as invocações a `EvalItem` podem decorrer em paralelo, o que seria vantajoso já que nessa operação se concentra a maior parte do tempo total de execução. Tirando partido da *Task Parallel Library*, apresente uma versão de `FindItem` que use invocações paralelas a `EvalItem` para tirar partido dos múltiplos *cores* de processamento disponíveis. Admita que qualquer elemento para o qual `EvalItem` retorne `true` é um resultado válido, independentemente da sua posição na sequência. Assegure-se de que a execução termina rapidamente após ser encontrado o item a retornar.

```
static Item FindItem(IEnumerable<Item> items, Query query) {
  foreach (Item item in items) { if (EvalItem(item, query)) return item; }
  return null;
}
```