Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Exame de Época Especial - parte escrita, Inverno de 2014/2015

---

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

```
public class UnsafeSpinBarrier {
  private int toArrive;

  public UnsafeSpinBarrier(int partners) {
    if (partners <= 0)
      throw new IllegalArgumentException();
    toArrive = partners;
  }

  public void signalAndWait() {
    if (toArrive == 0)
      throw new IllegalStateException();
    if (--toArrive > 0)
      do { Thread.currentThread().yield();
      } while (toArrive != 0);
  }

  public void addPartner() {
    if (toArrive == 0)
      throw new InvalidStateException();
    toArrive += 1;
  }

  public void removePartner() {
    if (toArrive > 0)
      toArrive -= 1
    else
      throw new InvalidStateException();
  }
}
```

   Esta classe sugere implementar o sincronizador *barrier* mas, como o próprio nome indica, não é *thread-safe*. Usando a linguagem *Java* e sem utilizar *locks*, implemente uma versão *thread-safe* deste sincronizador.

2. [4] Usando a linguagem *Java* e os respectivos monitores implemente o sincronizador *transient wait queue* com a classe `TransientWaitQueue` respeitando a seguinte interface pública:

```
public class TransientWaitQueue {
   public boolean await(long timeout) throws InterruptedException;
   public void signal();
   public void signalAll();
}
```

   A chamada ao método `await` bloqueia a *thread* invocante até que a mesma seja sinalizada (com os métodos `signal` ou `signalAll`), expire o tempo especificado através do argumento *timeout* ou seja interrompida a espera da *thread*. O método `signal` desbloqueia a *thread* que se encontre bloqueada há mais tempo (FIFO). O método `signalAll` liberta todas as *threads* que se encontram bloqueadas no sincronizador. Quando não houver *threads* bloqueadas, as invocações dos métodos `signal` e/ou `signalAll` não tem qualquer consequência. A implementação do sincronizador deve ter em consideração a possibilidade de interrupção das *threads*, enquanto estão bloqueadas, e a possibilidade do monitor gerar notificações espúrias.

3. [4] Implemente em *Java* ou C#, com base nos respectivos monitores implícitos ou explícitos, a classe `FairReadWriteLock` que implementa um sincronizador com uma semântica semelhante à do *read/write semaphore* no *kernel* do Linux. A interface pública da classe em C# é a seguinte:

```
public class FairReadWriteLock {
   public void EnterRead();
   public void EnterWrite();
   public void ExitRead();
   public void ExitWrite();
}
```

   Para que o sincronizador seja equitativo (*fair*) para os dois tipos de acessos (i.e., leitura e escrita) deve ser utilizada apenas uma fila de espera, com disciplina FIFO, onde são inseridos todos os pedidos de leitura ou escrita pendentes. O método `EnterRead` solicita o acesso para leitura ao recurso protegido pelo *lock*, solicitando o método `EnterWrite` o acesso para escrita. O acesso para leitura será concedido de imediato, quando a fila de espera estiver vazia e nenhuma outra *thread* possuir acesso para escrita. O acesso para escrita é concedido de imediato, quando a fila de espera estiver vazia e nenhuma outra *thread* possua qualquer tipo de acesso. Quando

o recurso protegido pelo *lock* deixar de ser acessível para leitura (i.e., a última *thread* leitora invoca o método `ExitRead`) ou para escrita (i.e., a thread com acesso para escrita invoca o método `ExitWrite`) e existirem pedidos em fila de espera, o sincronizador deve comportar-se do seguinte modo: (a) se o primeiro pedido da fila solicitar acesso para escrita, esse acesso será garantido à respectiva *thread*; (2) se o primeiro pedido da fila solicitar acesso para leitura, esse acesso será concedido à respectiva *thread*, assim como a todas as outras *threads* cujos respectivos pedidos de acesso para leitura se encontrarem imediatamente a seguir na fila de espera. A implementação deve ter em consideração a possibilidade de interrupção das *threads* enquanto estão bloqueadas e a possibilidade do monitor gerar notíficações espúrias.

4. [6] A interface `Operations` representa as acções usadas por um sistema de monitorização que avalia periodicamente qual de dois servidores está com melhor capacidade de resposta a pedidos. O método `CheckServers` invoca `TestRequest` para os dois servidores indicados e regista, com `SaveBest`, o endereço do servidor cuja resposta tem um valor mais alto.

```
public class Monitoring {
   public interface Operations {
      int TestRequest(Address server);
      void SaveBest(Address best);
   }
   public static void CheckServers(Operations ops, Address srv1, Address srv2) {
      int q1 = ops.TestRequest(srv1);
      int q2 = ops.TestRequest(srv2);
      ops.SaveBest(q1 > q2 ? srv1 : srv2);
   }
}
```

   a. [4] A classe `APMMonitoring` será a variante assíncrona de `Monitoring`, seguindo o estilo *Asynchronous Programming Model* (APM). Apresente os métodos `BeginCheckServers` e `EndChesckServers`, que usam a interface `APMOperations` (variante APM de `Operations` 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. [2] A classe `TAPMonitoring` será a variante assíncrona de `Monitoring`, seguindo o estilo *Task-based Asynchronous Pattern* (TAP). Tirando partido da *Task Parallel Library* (TPL) ou dos métodos `async` do C#, implemente o método `CheckServersAsync`, que usa a interface `TAPOperations` (variante TAP da interface `Operations` que não tem de apresentar).
   NOTA: na implementação não se admite a utilização de operações com bloqueios de controlo.

5. [3] Considere o método `Digest` apresentado a seguir. A sequência de invocações do método `Transform` não é paralelizável, já que cada invocação é parametrizada com o resultado da invocação anterior. No entanto, as invocações a `Process` podem decorrer em paralelo, o que se considera vantajoso já que é nessa operação que se concentra a maior parte do tempo total de execução. O método `Aggregate` implementa uma operação comutativa e associativa e `new Summary()` produz o seu elemento neutro. As funções `Transform` e `Process` só realizam operações de leitura sobre a instância de `Unit` que recebem como argumento. Tirando partido da *Task Parallel Library*, apresente uma versão de `Digest` que use invocações paralelas a `Process` para tirar partido de todos os *cores* de processamento disponíveis.

```
static Summary Digest(IEnumerable<Item> values, Info info) {
   Summary r, t = new Summary(); Unit u = new Unit();
   foreach (Item item in values) {
      u = Transform(u, item); r = Process(u, info); t = Aggregate(t, r);
   }
   return t;
}
```

Duração: 2 horas e 30 minutos
ISEL, 20 de Março de 2015