Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

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

---

1. [2.5] Considere a classe `UnsafeSpinReentrantLock`, cuja implementação, em *Java,* é apresentada a seguir:

```
class UnsafeSpinReentrantLock {
  private Thread owner;
  private int count;

  public boolean tryLock() {
    if (owner == Thread.currentThread()) { count++; return true; }
    if (owner == null) { owner = Thread.currentThread(); return true; }
    return false;
  }

  public void lock() { while (!tryLock()) Thread.yield(); }

  public void unlock() {
    if (owner != Thread.currentThread()) throw new IllegalMonitorStateException();
    if (count == 0) owner = null; else count--;
  }
}
```

   Esta implementação reflete a semântica do sincronizador *reentrant lock* disponível em *Java*, contudo não é *thread-safe*. Usando técnicas de sincronização *non-blocking*, implemente, em *Java* ou em C#, uma versão *thread-safe* deste sincronizador.

2. [4] Usando os monitores disponíveis nas linguagens C# ou *Java*, implemente o sincronizador *barrier*, cuja interface pública e semântica de sincronização se descrevem a seguir.

```
class Barrier {
  public Barrier(int participants);
  public boolean await(long timeout) throws InterruptedException;
}
```

   Este sincronizador destina-se a sincronizar grupos de *threads* cujo número de participantes é especificado na construção. As *threads* invocam o método **`await`** quando se pretendem sincronizar com as outras *threads* do grupo. A chamada e este método pode bloquear a *thread* invocante, e termina: (a) devolvendo **`true`**, após todas as *threads* do grupo terem invocado o método **`await`**; (b) devolvendo **`false`**, se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido. O método **`await`** só pode terminar por *timeout* ou interrupção se o *rendezvous* ainda não estiver concluído, isto é, faltar a invocação de **`await`** por pelo menos uma das *threads* do grupo. Do ponto de vista do estado do sincronizador, quando o uma *thread* desiste, por *timeout* ou interrupção, tudo se passa como se essa *thread* nunca tivesse invocado o método **`await`**. Por simplificação, considere que as instância desta classe são utilizadas apenas uma vez.

3. [4] Usando os monitores disponíveis na linguagem *Java* ou C#, Implemente o sincronizador *synchronization event*, cuja interface pública e a semântica de sicronizaçao se descrevem a seguir:

```
class SynchronizationEvent {
  public SynchronizationEvent(boolean signalled)
  public boolean await(long timeout) throws InterruptedException;
  public void set();
  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, deixando-o no estado não sinalizado; (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`

liberta a *thread* bloqueada há mais tempo no evento (FIFO) ou sinaliza o evento, no caso de não existirem *threads* bloqueada. A operação `reset` coloca o evento no estado não sinalizado. A implementação deste 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 **`Services`** define o conjunto de serviços síncronos disponibilizados por uma organização que oferece a execução de serviços nos seus servidores. O método **`Login`** estabelece uma sessão de utilização em proveito do utilizador especifcado, lançando a excepção **`IllegalUserException`**, quando as credenciais não são válidas. O método **`Execute`** executa, num servidor remoto, no âmbito da sessão especificada por **`session`**, o serviço **`service`** e devolve a respectiva resposta através do seu valor de retorno. O método **`LogUsage`**, que pode ser executado em paralelo com o método **`Execute`**, regista, num servidor de *logs*, o pedido de execução de um serviço. Por fim, o método **`Logout`** encerra a sessão iniciada com o método **`Login.`** O método estático **`ExecuteService`** executa sincronamente o serviço especificado através do parâmetro **`service`** em proveito do utilizador especificado com o parâmetro **`uid`**, usando as operações disponibilizadas por uma implementaçao da interface **`Services`**.

```
public class Execute {
  public interface Services {
    Session Login(UserID uid);
    void LogUsage(Session session, Request service);
    Response Execute(Session session, Request service);
    void Logout(Session session);
  }
  public static Response ExecuteService(Services svc, UserID uid, Request service) {
    Session session = svc.Login(uid);
    svc.LogUsage(session, service);
    Response response = svc.Execute(session, service);
    svc.Logout(session);
    return response;
  }
}
```

a) [3,5] A classe **`APMExecute`** será a variante assíncrona de **`Execute`** ao estilo *Asynchronous Programming Model* (APM). Implemente os métodos **`BeginExecuteService`** e **`EndExecuteService`** 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) [3,5] A classe **`TAPExecute`** será a variante assíncrona de **`Execute`**, 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 **`ExecuteServiceAsync`**, 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.

5. [2,5] O método **`MaxIndex`**, apresentado a seguir, determina o índice do *array* de inteiros, passado como argumento, onde se localiza o elemento com o maior valor. Tirando partido da *Task Parallel Library*, apresente uma versão do método **`MaxIndex`** de modo a tirar partido de todos os *cores* de processamento disponíveis.

```
static int MaxIndex(int[] data) {
  int index = 0;
  for (int i = 1; i < data.Length; i++) {
    if (data[i] > data[index]) index = i;
      if (data[index] == Int32.MaxValue) break;
  }
  return index;
}
```

Duração: 2 horas e 30 minutos
ISEL, 11 de Julho de 2016