Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 2ª Época,Verão de 2014/2015

---

1. [3] Considere a classe `UnsafeCLHLock`, como uma tentativa para implementar um *CLH queued spin lock* (CLH deriva do nome dos seus autores: *Craig*, *Landin*, e *Hagersten*).

```
public sealed class UnsafeCLHLock {
  public class CLHNode {
    internal bool succMustWait = true;  // The default is to wait for a lock
  }

  private CLHNode tail;    // the tail of wait queue; when null the lock is free

  public CLHNode Lock() {
    CLHNode myNode = new CLHNode();
    CLHNode predNode = tail; // insert myNode at tail of queue and get my predecessor
    tail = myNode;
    // If there is a predecessor spin until the lock is free; otherwise we got
    // the lock.
    if (predNode != null) {
       SpinWait sw = new SpinWait();
       while (predNode.succMustWait)
         sw.SpinOnce();
    }
    return myNode;
  }

  public void Unlock(CLHNode myNode) {
    // If we are the last node on the queue, then try to set tail to null.
    if (tail == myNode)
      tail = null;
    else
      myNode.succMustWait = false ;  // Grant access to the successor thread.
  }
}
```

   Esta classe ilustra o algoritmo do *CLH spin lock*, mas não é *thread-safe*. Usando apenas o atributo `volatile` e instruões atómicas, usando C# ou *Java*, implemente a classe *thread-safe* `CLHLock`, com a mesma semântica de sincronização e introduzindo as mesmas barreiras de memória que todos os *locks* introduzem, nos vários modelos de memória (e.g., JMM) (e.g., *acquire* na operação `Lock` e *release* na operação `Unlock`).

2. [5] Implemente, em *Java* ou C# usando monitores implícitos ou os monitores explícitos do *Java*, o sincronizador *Broadcast* que promove a difusão de mensagens do tipo genérico `M` (`Send`). O interesse na recepção das mensagens difundidas é manifestado através da operação `Receive`. A seguir, define-se em C# a interface pública deste sincronizado.

```
public class Broadcast<M> where M : class {
   public bool Send(M msg);
   public M Receive(int timeout);
}
```

   A operação `Send` entrega a mensagem (`msg`) a todas as *threads* em espera, retornando `true`, ou descarta a mensagem caso não existam *threads* em espera, retornando `false`. A operação `Receive` bloqueia a *thread* invocante até à recepção da próxima mensagem difundida, e pode terminar: com sucesso, retornando a mensagem difundida; produzindo `ThreadInterruptedException` no caso a *thread* tenha sido interrompida quando bloqueada, ou devolvendo `null` se for atingido o tempo máximo de espera especificado.

Nota: Lembre-se que as mensagens são apenas difundidas às *threads* que cujo interesse está registado (`Receive`) no momento da divulgação (`Send`).

3. [5] Implemente, em Java ou C# utilizando os monitores implícitos ou os monitores explíctos do *Java*, o sincronizador *SyncThreadPool* que promove a execução dos itens de trabalho submetidos em *worker threads*, criadas para esse efeito. O número de *worker threads* é passado como parâmetro de construção. A seguir, apresentamos a interface pública da classe que implementa este sincronizador em C#.

```
public class SyncThreadPool {
  public SyncThreadPool(int nworkerThreads);
  public void SyncQueueWorkItem(Action work);
}
```

A operação `SyncQueueWorkItem` submete para execução o trabalho passado como parâmetro (`work`) e bloqueia a *thread* invocante até que o trabalho submetido seja executado. O sincronizador tem que suportar correctamente a interrupção das *threads* em espera.

4. [7] A interface `Services` representa os serviços síncronos disponibilizados por uma organização que implementa diversos serviços usando servidores localizados em diferentes áreas geográficas. O método `PingServer` responde a um pedido de *ping*, devolvendo o respectivo `Uri`; o parâmetro do tipo `CancellationToken` (`ct`) deste método deve ser especificado com o valor `default(CancellationToken)`, quando se usa a versão APM dos serviços; na versão TAP, este prâmetro pode ser usado para cancelar operações de *ping* pendentes. O método `ExecuteService` executa o serviço especificado pelos tipos genéricos `S` (serviço) e `R` (resposta), no servidor especificado por parâmetro `server`. O método `ExecuteOnNearServer` usa as operações de `Services<S, R>` para executar de forma síncrona o serviço especificado, no servidor que primeiro responder ao serviço `PingServer` (que se considera ser o *near server*).

```
public class SyncOps<S, R> {
   public interface Services<_S, _R> {
     Uri PingServer(Uri server, CancellationToken ct);
      _R ExecuteService(Uri server, _S service);
   }
   public static R ExecuteOnNearServer(Services<S, R> svc, Uri[] servers,
                                       S service) { ... }
}
```

a) [4] A classe `APMOps<S,R>` será a variante assíncrona de `SyncOps<S,R>`, seguindo o estilo *Asynchronous Programming Model* (APM). Apresente os métodos `BeginExecuteOnNearServer` e `EndExecuteOnNearServer`, que usam a interface `APMServices<_S, _R>` (variante APM de `Services<S,R>` 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<S,R>` será a variante assíncrona de `SyncOps<S,R>`, seguindo o estilo *Task-based Asynchronous Pattern* (TAP). Tirando partido da *Task Parallel Library* (TPL) ou dos métodos `async` do C#, implemente o método `ExecuteOnNearServerAsync`, que usa a interface `TAPServices<_S,_R>` (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, 18 de Julho de 2015