Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 1ª Época, Verão de 2016/2017

---

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

```
public class UnsafeRefCountedHolder<T> where T : class {
  private T value; private int refCount;

  public UnsafeRefCountedHolder(T v) { value = v; refCount = 1; }

  public void AddRef() {
    if (refCount == 0) throw new InvalidOperationException();
    refCount++;
  }

  public void ReleaseRef() {
    if (refCount == 0) throw new InvalidOperationException();
    if (--refCount == 0) {
      IDisposable disposable = value as IDisposable; value = null;
      if (disposable != null) disposable.Dispose();
    }
  }

  public T Value {
    get {
      if (refCount == 0) throw new InvalidOperationException();
      return value;
    }
}
```

   Esta classe implementa um tipo de dados destinado a armazenar objectos partilhados entre *threads* com o tempo de vida gerido com base na contagem de referências, contudo não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, uma versão *thread-safe* deste tipo.

2. [3,5] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *notification event*, cuja interface pública em *Java* é a seguinte:

```
public class NotificationEvent {
  public NotificationEvent(boolean signaled);
  public boolean await(long timeout) throws InterruptedException;
  public void set();
  public void pulse();
  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 e não modifica o estado do evento; (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`** sinalza o evento libertando todas as *threads* nele bloqueadas. A operação **`pulse`** liberta eventuais *threads* bloqueadas no evento deixando-o no estado não sinalizado. A operação **`reset`** coloca o evento no estado não sinalizado.

3. [3.5] Implemente em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *keyed exchanger* que permite a troca de dados entre pares de *threads* identificados por uma chave. A interface pública deste sincronizador em C# é a seguinte:

```
public class KeyedExchanger<T> {
  public bool Exchange(int pairKey, T myData, int timeout, out T yourData);
}
```

As *threads* que utilizam este sincronizador manifestam a sua disponibilidade para iniciar uma troca invocando o método **Exchange**, especificando a identificação do par (**pairKey**) , o objecto que pretendem entregar à *thread* parceira (**myData**) e, opcionalmente, o tempo limite da espera pela troca (**timeout**). O método **Exchange** termina: (a) devolvendo **true**, quando é concluída a troca com outra *thread*, sendo os dados por ela oferecidos retornado através do parâmetro **yourData**; (b) devolvendo **false**, se expirar o limite do tempo de espera especificado, ou; (c) lançando **ThreadInterruptedException** quando a espera da *thread* é interrompida. A implementação deste sincronizador deve procurar minimizar o número de comutações de *threads* em todas as circunstâncias, usando as técnicas estudadas na unidade curricular.

Nota: Se implementar o sincronizador em *Java* altere adequadamente a assinatura do método **Exchange**.

4. [8] A interface **Services** define os serviços síncronos disponibilizados por uma organização que oferece a execução de serviços nos seus servidores mediante o pagamento de uma taxa. O método **PingServer** permite nterrogar cada servidor para obter informação sobre o respectivo custo de utilização assim como do URI que disponibilza o serviço. Este método permite cancelamento, através do parâmetro **ctoken**, e lança a excepção **UnavailableServerException** sempre que o servidor não se encontra disponível. O método **ExecService** executa, no servidor especificado através do parâmetro **server**, o serviço especificado pelo argumento **service** e devolve, por valor, o respectivo resultado**.** O método **ExecOnCheaperThanServer** usa as operações de **Services** para executar de forma síncrona o serviço especificado, no primeiro servidor a responder que apresentar um custo inferior ao especificado com o argumento **thresholdCost**. Se não nenhum dos servidores especificados com o argumento **servers** estiver disponível ao custo pretendido, o método **ExecOnCheaperThanServer** deve lançar a excepção **UnavailableServerException**; em todas as circunstâncias, antes de retornar, este método deverá cancelar as operações de interrogação pendentes.

```
public class Exec {
  public class ServerInfo { public long cost; public Uri serviceUri; }
  public interface Services {
    ServerInfo PingServer(Uri server, CancelationToken ctoken);
    ResponseType ExecService(Uri server, ServiceType service);
  }
  public ResponseType ExecOnCheaperThanServer(Services svc, Uri[] servers,
                                              ServiceType service, long thresholdCost);
}
```

   a. [4] A classe **APMExec** será a variante assíncrona de **Exec** ao estilo *Asynchronous Programming Model* (APM). Implemente os métodos **BeginExecOnCheaperThanServer** e **EndExecOnCheaperThanServer** 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. [4] 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 **ExecOnCheaperThanServerAsync**, 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 **FindMaxIndex**, apresentado a seguir, determina o índice do *array*, passado como argumento, onde se localiza o elemento com maior valor. Tirando partido da *Task Parallel Library*, apresente uma versão do método **FindMaxIndex** de modo a tirar partido de todos os *cores* de processamento disponíveis. O método suporta cancelamento através do parâmetro **ctoken**.

```
public static int FindMaxIndex(long[] values, CancellationToken ctoken) {
  int index = 0;
  for (int i = 1; i < values.Length; i++) {
    if (values[i] > values[index]) index = i;
    if (values[index] == Int64.MaxValue) break;
  }
  return index;
}
```

Duração: 2 horas e 30 minutos
ISEL, 3 de Julho de 2017