Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 2ª Época, Inverno de 2017/2018

---

1. [2,5] Considere a definição em C# da classe `UnsafeWindowsCriticalSection` como uma tentativa para implementar o sincronizador *critical section*.

```
public class UnsafeWindowsCriticalSection {
  private int state = -1;      // -1 when the CS is free
  private int owner;           // identifier of the CS' owner thread
  private int enterCount;      // number of times the CS was entered by the current owner
  private EventWaitHandle waitEvent; // lazy created, event where waiters are blocked

  private EventWaitHandle WaitEvent {
    get {
      if (waitEvent == null) waitEvent = new AutoResetEvent(false);
      return waitEvent;
    }
  }

  public void Enter() {
    int tid = Thread.CurrentThread.ManagedThreadId;
    if (++state == 0) {  // CS was free, we acquired it
      owner = tid; enterCount = 1;
    } else if (owner == tid) enterCount++;  // recursive enter
    else { // the CS is owned by another thread, so the current thread must wait
      WaitEvent.WaitOne();
      owner = tid; enterCount = 1;
    }
  }

  public void Leave() {
    if (owner != Thread.CurrentThread.ManagedThreadId)
      throw new SynchronizationLockException();
    if (--enterCount > 0) // recursive leave
      state -= 1;
    else { // last leave: clear the owner thread
      owner = 0;
      if (--state >= 0) // At least one thread is waiting, wake one of the waiting threads
        WaitEvent.Set();
    }
  }
}
```

   Ainda que esta classe siga o algoritmo usado no sistema operativo *Windows*, não é *thread safe*. Sem usar *locks*, implemente, em C# ou *Java*, uma versão *thread safe* deste sincronizador. (Se optar por usar *Java*, utilize o sincronizador `java.util.concurrent.Semaphore` para bloquear as *threads* na *critical section*).

2. [4] Implemente, em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *block messsage queue*, que permite a comunicação entre *threads* produtoras e *threads* consumidoras através de mensagens podem ser enviadas e recebidas em bloco, e cuja interface pública em *Java* é a seguinte:

```
public class BlockMessageQueue<T> {
  public void putBlock(T[] msgBlock);
  public T[] takeBlock(int blockSize, int timeout) throws InterruptedException;
}
```

   A operação **`putBlock`** é usada pelas *threads* produtoras para entregar à fila as mensagens especificadas com o argumento **`msgBlock`** e nunca bloqueia a *thread* invocante. A operação **`takeBlock`** permite às *threads* consumidoras receberem um bloco de mensagens, com a dimensão especificada por **`blockSize`**, e termina:

(a) com sucesso, devolvendo o *array* com o bloco de mensagens recebidas; (b) devolvendo **`null`** se expirar o limite de tempo especificado para tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido. A ordem das mensagens recebidas em cada bloco devem respeitar a ordem pela qual as mesmas foram enviadas e a implementação do sincronizador deverá optimizar o número de comutações de contexto.

3. [4] Implemente, em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *keyed event*, implementado no sistema operativo *Windows*, cuja interface pública em *Java* é a seguinte:

```
public class WindowsKeyedEvent {
    public boolean release(Object key, int timeout) throws InterruptedException;
    public boolean await(Object key, int timeout) throws InterruptedException;
}
```

   As operações **`release`** e **`await`** são ambas bloqueantes; a operação **`release`** bloqueia a *thread* invocante até que seja invocada, por outra *thread*, a operação **`await`** especificando a mesma chave; o simétrico acontece com a operação **`await`**. As operações **`release`** e **`await`** podem bloquear a *thread* invocante, e terminam: (a) devolvendo **`true`** se ocorrer a chamada à operação complementar; (b) devolvendo **`false`** se expirar o limite especificado para o tempo de espera, ou; (c) lançando **`InterruptedException`**, se o bloqueio da *thread* for interrompido. As *threads* devem ser desbloqueadas com disciplina *first-in-first-out* (FIFO), isto é, a operação **`release`** deve desbloquear a *thread* que se encontra bloqueada há mais tempo por **`await`** e a operação **`await`** deverá desbloquear a *thread* que se encontra bloqueada há mais tempo por **`release`**.

4. [7] As operações **`Map`** e **`Join`** não têm efeitos colaterais pelo que podem ser executadas em paralelo.

```
public class Question4 {
  public static R Map(T t);
  public static R Join(R r, R r2);

  public static R[] MapJoin(T[] items) {
    var res = new R[items.Length / 2];
    for(int i = 0 ; i < res.Length; i++) {
      res[i] = Join(Map(items[2 * i]), Map(items[2 * i + 1]));
    return res;
  }
}
```

   a. [3,5] Realize uma versão assíncrona do método **`MapJoin`** usando o padrão APM (*Asynchronous Programming Model*), assumindo que tem disponível versões APM dos métodos **`Map`** e **`Join`**. Tire partido do paralelismo potencial existente.

   b. [3,5] Realize uma versão assíncrona do método **`MapJoin`** seguindo o padrão TAP (*Task-based Asynchronous Pattern*) usando a TPL e/ou os métodos assíncronos do C#. Assuma que tem disponível versões TAP dos métodos **`Map`** e **`Join`**. Tire partido do paralelismo potencial existente e se utilizar os métodos assíncronos do C# considere o recursos a métodos auxiliares.

5. [2,5] Usando os mecanismos disponíveis na *Task Parallel Library* de forma a utilizar todos os *cores* de processamento disponíveis, implemente o método **`AtLeastOccursParallel`** que determina se pelo menos **`occurrences`** elementos do enumerável **`items`** obedecem ao predicado especificado por **`selector`**. Considere que as chamadas ao predicado podem ser feitas em paralelo, que a operação é cancelável através do parâmetro **`ctoken`** e que o método deve retornar o mais rapidamente possível após ser determinado o resultado.

```
public static bool AtLeastOccursParallel<T>(IEnumerable<T> items, Predicate<T> selector,
                                            int occurrences, CancellationToken ctoken);
```

Duração: 2 horas e 30 minutos
ISEL, 22 de Janeiro de 2018