Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 2ª Época, Verão de 2017/2018, 12 de Julho de 2018

---

1. [2,5] Considere a definição em C# da classe `UnsafeReadWriteSpinLock` como uma tentativa para implementar o sincronizador *read/write spin lock*.

```
public class UnsafeReadWriteSpinLock {
  private const int UNLOCKED = 1 << 24;
  private int state = UNLOCKED;  // UNLOCKED when there are no active readers or writers
  public void LockRead() {
    do {
      if (--state >= 0) return;
      state++;      // undo previous decrement
      SpinWait sw = new SpinWait();
      do { sw.SpinOnce(); } while (state < 1);
    } while (true);
  }
  public void UnlockRead() { state++; }
  public void LockWrite() {
    do {
      if ((state -= UNLOCKED) == 0) return;
      state += UNLOCKED;      // undo previous subtraction
      SpinWait sw = new SpinWait();
      do { sw.SpinOnce(); } while (state != UNLOCKED);
    } while (true);
  }
  public void UnlockWrite() { state += UNLOCKED; }
}
```

   Ainda que o algoritmo seja o utilizado pelo sistema operativo *Linux*, esta implementação não é *thread safe*. Sem usar *locks*, implemente, em C# ou *Java*, uma versão *thread safe* deste sincronizador.

2. [4] Implemente, em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *simple barrier*, cuja interface pública, em *Java*, se apresenta a seguir.

```
class SimpleBarrier {
  public SimpleBarrier(int participants);
  public boolean await(long timeoutMillis) 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 mesmo grupo. A chamada a 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 a sincronização ainda não estiver concluída, 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 deve passar como se essa *thread* nunca tivesse invocado o método **`await`**. Por simplificação, considere que as instância desta classe são utilizadas apenas uma vez.

3. [4,5] Implemente, em *Java* ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *read/write semaphore* cuja interface é semelhante à disponível no *kernel* do sistema operativo *Linux*. A interface pública deste sincronizador, definida em C#, é a seguinte:

```
public class RwSemaphore {
  public void DownRead();
  public void DownWrite();
  public void UpRead();
  public void UpWrite();
```

```
  public void DowngradeWriter();
}
```

Os métodos **DownRead** e **DownWrite** adquirem a posse do semáforo para leitura ou escrita, respectivamente. Os métodos **UpRead** e **UpWrite** libertam o semáforo depois da *thread* invocante o ter adquirido para leitura ou para escrita, respectivamente. O método **DowngradeWriter**, invocado pelas *threads* que tenham previamente adquirido o semáforo para escrita, liberta o acesso para escrita e, atomicamente, adquire acesso para leitura.
Para que o semáforo seja equitativo na atribuição dos dois tipos de acesso (leitura e escrita), deverá ser utilizada uma única fila de espera, com disciplina FIFO, para garantir que as solicitações de aquisição (leitura e escrita) sejam servidas por ordem de chegada.
O acesso para leitura deve ser concedido: (a) quando solicitado (**DownRead**), se a fila de espera estiver vazia e não existir escrita em curso, ou; (b) quando termina uma escrita (**UpWrite**), ao grupo de *threads* leitoras que eventualmente se encontrem no início da fila de espera. O acesso para escrita deve ser concedido: (a) quando solicitado (**DownWrite**), se a fila de espera estiver vazia e não existir escrita nem leitura(s) em curso, ou; (b) quando termina a última leitura (**UpRead**) de um grupo de leitores, à *thread* escritora que eventualmente se encontre à cabeça da fila de espera. (Por exemplo, se a fila de espera tiver pendentes os pedidos W1, R1, R2 e W2 a ordem de atendimento será: escrita W1, leitura R1 e R2 em simultâneo e, finalmente, escrita W2.)
A implementação deve suportar o cancelamento dos métodos bloqueantes quando as *threads* bloqueadas são interrompidas (lançando **ThreadInterruptedException**) e deve optimizar o número de comutações de *thread* que ocorrem nas diferentes transições de estado.

4. [6] A interface **ITAPServices** define os serviços assíncronos disponibilizados por uma empresa que permite aos seus clientes a possibilidade de executar processamento nos seus servidores. O método **LoginAsync** permite estabelecer uma sessão de utilização em proveito do utilizador especifcado por **uid**. O método **ExecServiceAsync** executa um serviço no âmbito de uma sessão e pode ser invocado em paralelo para executar vários serviços. O método estático **ExecServicesAsync** é uma tentativa para implementar a execução assíncrona do conjunto de serviços.

```
public interface ITAPServices {
  Task<Session> LoginAsync(UserID uid);
  Task<Response> ExecServiceAsync(Session session, Request request);
  Task LogoutAsync(Session session);
}
```

```
public static Task<Response[]> ExecServicesAsync(ITAPServices svc, UserID uid,
                                                 Request[] requests) {
  return Task.Run(() => {
    Session session = svc.LoginAsync(uid).Result;
    try { Response[] responses = new Response[requests.Length];
      for (int i = 0; i < requests.Length; i++)
        responses[i] = svc.ExecServiceAsync(session, requests[i]).Result;
      return responses;
    } finally { try { svc.LogoutAsync(session).Wait(); } catch {} }
  });
}
```

   a. [2] Diga porque razões o método **ExecServicesAsync** não explora o paralelismo potencial expresso no algoritmo e também não optimiza a utilização dos recursos computacionais disponíveis.

   b. [4] Usando a TPL e/ou os métodos assíncronos do C#, faça uma implementação do método **ExecServicesAsync** de modo a explorar o paralelismo potencial e optimizar a utilização dos recursos computacionais disponíveis.

5. [3] Usando os mecanismos disponíveis na *Task Parallel Library*, implemente uma versão do método estático **ParallelMapSelected** que tire partido de todos os *cores* de processamento disponíveis no computador. Este método retorna o resultado da transformação (*mapping*) de todos os elementos de um enumerado que satisfaçam o predicado especificado. Tenha em consideração que o processamento dos elementos do enumerado pode ser executado em paralelo, que a operação de transformação é aquela que concentra a maior componente de processamento, que a operação do método deve poder ser cancelada e que a ordem dos elementos na lista resultante não é relevante.

```
public static List<O> ParallelMapSelected<I,O>(IEnumerable<I> items,
                      Predicate<I> selector, Func<I,O> mapper, CancellationToken ctoken);
```

Duração: 2 horas e 30 minutos