Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Teste Global de 1ª Época, Inverno de 2015/2016

---

1. [3] Considere a classe `UnsafeSpinLazy<T>` cuja implementação em C# é apresentada a seguir:

```
public class UnsafeSpinLazy<T> where T: class {
  private const int UNCREATED = 0, BEING_CREATED = 1, CREATED = 2;
  private int state = UNCREATED;
  private Func<T> factory;
  private T value;

  public UnsafeSpinLazy(Func<T> factory) { this.factory = factory; }

  public bool IsValueCreated { get { return state == CREATED; } }

  public T Value {
    get { SpinWait sw = new SpinWait();
      do {
        if (state == CREATED) {
          break;
        } else if (state == UNCREATED) {
          state = BEING_CREATED; value = factory(); state = CREATED; break;
        }
        sw.SpinOnce();
      } while (true);
      return value;
    }
  }
}
```

   A implementação deste sincronizador, cuja semântica de sincronização é idêntica à do tipo `Lazy<T>` do *.NET Framework*, não é *thread-safe*. Sem utilizar *locks*, implemente uma versão *thread-safe* deste sincronizador.

2. [3] Implemente em Java ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *channel with priority*, para suportar a comunicação entre *threads* usando mensagens. A interface pública deste sincronizador é a seguinte:

```
public class ChannelWithPriority<T> {
   public void Put(T msg, bool urgent);
   public bool Take(int timeout, out T rcvdMsg);
}
```

   O método `Put` entrega uma mensagem ao canal, sendo a sua prioridade definida com o argumento `urgent`. Invocando o método `Take`, a respectiva *thread* manifesta a intenção de receber uma mensagem, ficando bloqueada se não existir nenhuma mensagem disponível. O método `Take` termina: retornando `true`, quando é recebida uma mensagem, sendo esta devolvida através do parâmetro `rcvdMsg`; retornando `false`, se expirar o limite do tempo de espera, ou; lançando `ThreadInterruptedException` quando a espera da *thread* é interrompida. As mensagens deve ser entregues às *threads* consumidoras tendo em consideração a sua prioridade (urgente ou normal) e para cada prioridade a ordem com que foram entregues ao canal (FIFO). As *threads* consumidoras devem ser servidas pela ordem inversa da invocação do método `Take` (LIFO).

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

3. [4] Implemente em Java ou C#, com base nos monitores implícitos ou explícitos, o sincronizador *rendezvous*, para sincronizar pares de *threads* identificados por uma chave; subjacente a cada acto de sincronização está também a troca de dados entre as duas *threads*. A interface publica deste sincronizador é a seguinte:

```
public class Rendezvous<T> {
   public bool DoIt(int rvkey, T mydata, int timeout, out T yourData);
}
```

(cont.)

As *threads* que utilizam este sincronizador manifestam a sua disponibilidade para iniciar um *rendezvous* invocando o método `DoIt`, especificando a identificação do *rendezvous* (`rvkey`), o objecto que pretendem entregar à *thread* parceira (`myData`) e, opcionalmente, o tempo limite da espera pelo *rendezvous* (`timeout`). O método `DoIt` termina: devolvendo `true`, quando é concluído o *rendezvous* com outra *thread*, sendo os dados por ela oferecidos retornado através do parâmetro `yourData`; devolvendo `false`, se expirar o limite do tempo de espera especificado, ou; lançando `ThreadInterruptedException` quando a espera da *thread* é interrompida.
Nota: Se implementar o sincronizador em Java altere adequadamente a assinatura do método `DoIt`.

4. [7] A interface `Users.Service` disponibiliza serviços base de um sistema de gestão de utilizadores, apenas em versões assíncronas, quer no modelo *Asynchronous Programming Model* (APM), quer no modelo *Task-based Asynchronous Pattern* (TAP). A classe `Users` disponibiliza ainda a operação composta `GetUserAvatarAsync`, também assíncrona, que invoca dois serviços assíncronos em sequência. No entanto, verifica-se que o uso desta operação resulta num consumo considerável de recursos nos servidores do sistema, principalmente nos momentos em que há mais pedidos em curso simultaneamente.

```
public class Users {
  public interface Service {
    IAsyncResult BeginFindId(String name, String bdate, AsyncCallback cb, Object stt);
    int EndFindId(IAsyncResult asyncRes);
    Task<int> FindIdAsync(String name, String birthdate);
    IAsyncResult BeginObtainAvatarUri(int userId, AsyncCallback cb, Object stt);
    Uri EndObtainAvatarUri(IAsyncResult asyncRes);
    Task<Uri> ObtainAvatarUriAsync(int userId);
  }
  public static Task<Uri> GetUserAvatarAsync(Service svc, String name, String bdate) {
    return Task.Run(() => {
      int userId = svc.FindIdAsync(name, bdate).Result;
      return svc.ObtainAvatarUriAsync(userId).Result;
    });
  }
}
```

   a. [1.5] Sabendo que as operações de `Users.Service` consistem essencialmente em I/O, qual é o defeito grave de implementação de `GetUserAvatarAsync` e porque razão este impõe um elevado consumo de recursos quando é intensivamente utilizado em vários pedidos em simultâneo?
   b. [2] Apresente uma implementação corrigida de `GetUserAvatarAsync`, utilizando devidamente a *Task Parallel Library* (TPL) e/ou os métodos *async* de C#.
   c. [3.5] Acrescente à classe Users as operações `BeginGetUserAvatar` e `EndGetUserAvatar`, para que a operação `GetUserAvatar` também possa ser utilizada de acordo com o modelo APM.
      NOTA: não pode usar a TPL e só se admitem esperas de controlo dentro da operação `End`, estritamente onde o APM o exige.

5. [3] No método `ProcessItems`, apresentado a seguir, as invocações a `ExtractInfo` podem decorrer em paralelo, o que seria vantajoso já que nessa operação se concentra a maior parte do tempo total de execução. O método `MergeInfo` implementa uma operação comutativa e associativa e `new Info()` produz o seu elemento neutro. A função `Info ExtractInfo(...)` só realiza operações de leitura sobre a instância de `Data` que recebe como argumento. Tirando partido da *Task Parallel Library*, apresente uma versão de `ProcessItems` que use invocações paralelas a `ExtractInfo` para tirar partido de todos os *cores* de processamento disponíveis.

```
static Info ProcessItems(Data[] items, Session session) {
  Info info = new Info();
  for (int i = 0; i < items.Length; ++i)
    info = MergeInfo(info, ExtractInfo(items[i], session));
  return info;
}
```

Duração: 2 horas e 30 minutos
ISEL, 22 de Janeiro de 2016