Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente, Verão de 2016/2017

Segunda série de exercícios

Resolva os seguintes exercícios e apresente os programas de teste com os quais validou a correção da implementação de cada exercício.

1. Implemente em Java e C# a classe **`ConcurrentQueue`**<T> que define um contentor com disciplina FIFO (*First In First Out*) suportado numa lista simplesmente ligada. A classe disponibiliza as operações **`put`**, **`tryTake`** e **`isEmpty`**. A operação **`put`** coloca no fim da fila o elemento passado como argumento; a operação **`tryTake`** retorna o elemento presente no início da fila ou **`null`**, no caso da fila estar vazia; a operação **`isEmtpy`** indica se a fila está vazia. A operação suporta acessos concorrentes e nenhuma das operações disponibilizadas bloqueia as threads invocantes.
   Nota: Na implementação tenha em consideração as explicações sobre *lock-free queue*, proposta por *Michael e Scott*, que consta no Capítulo 15 do livro J*ava Concurrency in Practice*.

2. 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* desta classe.

3. Tirando partido dos mecanismos *non-blocking* discutidos nas aulas teóricas, implemente em C# uma versão optimizada do sincronizador **`ExpirableLazy`** (cuja especificação consta na primeira série de exercícios). As optimizações devem incidir sobre as seguintes situações: o valor já se encontra disponível; o valor não se encontra disponível e não existe outra *thread* a calculá-lo.

4. [**Opcional**] No artigo Nonblocking Concurrent Data Structures with Condition Synchronization, *William N. Scherer III* e *Michael L. Scott* propõem duas estruturas de dados *lock free*,para utilizar na comunicação de dados entre *threads*, designadas pelos autores por *dual stack* e *dual queue*. Os algoritmos propostos no artigo encontram-se aqui descritos em pseudocódigo.
   Tendo em consideração o artigo citado acima, o pseudocódigo associado ao artigo e o código distribuído no anexo, complete a implementação, em *Java*, da classe **`LockFreeDualQueue<T>`**. Esta classe define uma *dual data queue*, que se destina a suportar comunicação entre *threads*, em cenários produtor/consumidor, onde a espera em ciclo *busy-wait* seja adequada. A classe a implementar deve disponibilizar as operações **`enqueue`**, **`dequeue`** e **`isEmpty`**. A operação **`enqueue`** coloca no fim da fila o elemento passado como argumento, satisfazendo uma operação **`dequeue`** pendente, se existir; a operação **`dequeue`** retorna o item de dados mais antigo que se encontra na fila, forçando a *thread* invocante a espera enquanto a fila estiver vazia; a operação **isEmpty** indica se a fila se encontra vazia ou se apenas contém nós inseridos pela operação **`dequeue`** (nós do tipo *request*).

Data limite de entrega: 28 de Novembro de 2017

ISEL, 31 de Outubro de 2017