

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

1. Considere a classe **UnsafeMessageBox**, cuja implementação em C# se apresenta a seguir:

```
public class UnsafeMessageBox<M> where M : class {  
    private class MsgHolder {  
        internal readonly M msg;  
        internal int lives;  
    }  
  
    private MsgHolder msgHolder = null;  
    public void Publish(M m, int lvs) {  
        msgHolder = new MsgHolder { msg = m, lives = lvs };  
    }  
  
    public M TryConsume() {  
        if (msgHolder != null && msgHolder.lives > 0) {  
            msgHolder.lives -= 1;  
            return msgHolder.msg;  
        }  
        return null;  
    }  
}
```

Esta implementação reflete a semântica de uma *message box* contendo no máximo uma mensagem que pode ser consumida múltiplas vezes, contudo não é *thread-safe*. Implemente em *Java* ou em *C#*, sem utilizar *locks*, uma versão *thread-safe* deste sincronizador.

2. Tirando partido dos mecanismos *non-blocking* discutidos nas aulas teóricas, implemente em *Java* uma variante otimizada do sincronizador **MessageQueue**, cuja especificação original consta na primeira série de exercícios. Esta variante apresenta as seguintes diferenças em relação a essa versão original: (a) a interface **SendStatus** deixa de suportar o método **tryCancel**; (b) a obtenção de mensagens por parte das *threads* consumidores não têm de seguir a ordem FIFO (*first in first out*), contudo a entrega das mensagens continua a seguir essa ordem. As optimizações devem incidir sobre as seguintes situações: (a) no envio de mensagem, quando não existe nenhuma *thread* à espera da mensagem; (b) na recepção de mensagem, quando já existem mensagens em fila.

Nota: Na implementação tenha em consideração as explicações sobre a *lock-free queue*, proposta por Michael e Scott, que consta no Capítulo 15 do livro *Java Concurrency in Practice*.

3. [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: 26 de Novembro de 2018

ISEL, 31 de Outubro de 2018