Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente

Verão de 2020/2021, 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. A entrega deve ser feita através da criação da tag **`0.2.0`** no repositório individual de cada aluno.

1. Considere a definição em *Java* da classe **`UnsafeCountedLazy`** como uma tentativa para implementar um contentor que calcula o valor contido apenas aquando da primeira chamada de **`acquire`**, usando a função **`supplier`**. Em adição, este contentor chama a função **`closer`** sobre o valor, quando já não existirem mais utilizações.
   Contudo esta classe não é *thread-safe*. Sem usar *locks*, implemente uma versão *thread-safe*.

```
public class UnsafeCountedLazy<T> {
   private final Supplier<T> supplier;
   private final Consumer<T> closer;
   private int counter = -1;
   private T value = null;

   public UnsafeCountedLazy(Supplier<T> supplier, Consumer<T> closer) {
       this.supplier = supplier;
       this.closer = closer;
   }

   public T acquire() {
       if (counter == 0) throw new IllegalStateException("Object is closed");
       if (counter == -1) {
           counter = 1;
           value = supplier.get();
       } else {
           counter += 1;
           while (value == null) Thread.yield();
       }
       return value;
   }

   public void release() {
       if (counter <= 0) throw new IllegalStateException();
       if (--counter == 0) {
           closer.accept(value);
       }
   }
}
```

2. 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 um sincronizador *message box* contendo no máximo uma mensagem que pode ser consumida múltiplas vezes, até ao máximo de **`lvs`**. Contudo esta classe não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, uma versão *thread-safe* deste sincronizador.

3. Implemente o sincronizador *message queue*, para suportar a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico **`E`**. A entrega de mensagens deve usar o critério FIFO (*first in first out*): dadas duas mensagens colocadas na fila, a primeira a ser entregue a um consumidor deve ser a primeira que foi colocada na fila. Contudo esse critério FIFO não tem de ser garantido nas *threads* consumidores: um pedido de remoção pode ser satisfeito antes de outro pedido de remoção realizado previamente. A interface pública deste sincronizador, em *Java*, é a seguinte:

```
public class MessageQueue<E> {
  public void enqueue(E message);
  public Optional<E> dequeue(long timeout) throws InterruptedException;
}
```

O método **enqueue** não é bloqueante, retornando imediatamente após entregar a mensagem na fila. Optimize este sincronizador, usando as técnicas *non-blocking* apresentadas nas aulas teóricas. Note que o método **dequeue** continua a ser potencialmente bloqueante.

Data limite de entrega: 22 de maio de 2021

ISEL, 1 de maio de 2021