Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Programação Concorrente, Inverno de 2019/2020

Série de Exercícios 2

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 **`UnsafeBoundedLazy<E>`**, cuja implementação em *Java* se apresenta a seguir:

```
class UnsafeBoundedLazy<E> {
    private static class ValueHolder<V> {
        V value;
        int availableLives;

        ValueHolder(V value, int lives) {
            this.value = value;
            availableLives = lives;
        }

        ValueHolder() {}
    }

    // Configuration arguments
    private final Supplier<E> supplier;
    private final int lives;

    /**
     * The possible states:
     *   null: means UNCREATED
     *   CREATING and ERROR: mean exactly that
     *   != null && != ERROR && != CREATING: means CREATED
     */
    private final ValueHolder<E> ERROR = new ValueHolder<>();
    private final ValueHolder<E> CREATING = new ValueHolder<>();

    // The current state
    private ValueHolder<E> state = null;

    // When the synchronizer is in ERROR state, the exception is hold here
    Throwable errorException;

    // Construct a BoundedLazy
    public UnsafeBoundedLazy(Supplier<E> supplier, int lives) {
        if (lives < 1)
            throw new IllegalArgumentException();
        this.supplier = supplier;
        this.lives = lives;
    }
```

```
    // Returns an instance of the underlying type
    public Optional<E> get() throws Throwable {
        while (true) {
            if (state == ERROR)
                throw errorException;
            if (state == null) {
                state = CREATING;
                try {
                    E value = supplier.get();
                    if (lives > 1) {
                        state = new ValueHolder<E>(value, lives - 1);  //lives remaining
                    } else {
                        state = null;   // the unique live was consumed
                    }
                    return Optional.of(value);
                } catch (Throwable ex) {
                    errorException = ex;
                    state = ERROR;
                    throw ex;
                }
            } else if (state == CREATING) {
                do {
                    Thread.yield();
                } while (state == CREATING);      // spin until state != CREATING
            } else { // state is CREATED: we have at least one life
                Optional<E> retValue = Optional.of(state.value);
                if (--state.availableLives == 0)
                    state = null;
                return retValue;
            }
        }
    }
}
```

Esta implementação reflete a semântica duma variante do sincronizador **`Lazy<T>`**, em que o valor calculado pode ser disponibilizado no máximo **`lives`** vezes, antes de ter de ser recalculado. Contudo não é *thread-safe*. Implemente em *Java* ou em C#, sem utilizar *locks*, a classe **`SafeBoundedLazy<E>`** uma versão *thread-safe* deste sincronizador.

2. Tirando partido da sincronização *non-blocking* e das otimizações possíveis nas implementações de sincronizadores com base em monitor discutidos nas aulas teóricas, implemente em *Java* ou C# uma variante optimizada do sincronizador **`TransferQueue`**, cuja especificação original consta na primeira série de exercícios. Esta variante apresenta as seguintes diferenças em relação a essa versão original: (a) apenas suporta os métodos **`put`** e **`take`**; (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 otimizações devem incidir sobre o *fast-path* das operações **`put`** e **`take`**, nomeadamente: (a) o envio de uma mensagem, quando não existe nenhuma *thread* bloqueada pelo método **`take`**; (b) na recepção de mensagem, quando já existem mensagens disponíveis para recepção.

   Nota: Na implementação deve utilizar a *lock-free queue*, proposta por *Michael e Scott*, cuja explicação consta na Secção 15.4.2 do livro J*ava Concurrency in Practice*

Data limite de entrega: 7 de Junho de 2020.

ISEL, 18 de Maio de 2020