

**Programação Concorrente**

Semestre Inverno

LI52D

2017/2018

Engenheiro Pedro Félix

**Série de Exercícios 2**

42208 Diogo Dias

**Exercício 1**

Neste exercício foi pedido a implementação de uma Classe ConcurrentQueue<T> que define um contentor com disciplina FIFO (First In First Out). Esta classe tem que ser thread safe e seja free-lock. Esta classe é composta por três métodos:

void put(T item) – Este método coloca o objeto recebido por parâmetro na fila. A lógica deste método é o mesmo referido no capitulo 15 do livro Java Concurrency in Practice.

T tryTake() – Este método retorna o objeto que se encontra a mais tempo na fila, caso haja elementos. Se não houver elementos este retorna null. Na implementação deste método é alterado só a referencia do head. Quando for necessário remover um elemento, e houver elementos, é alterado a referencia do head para head.next. Com esta mudança o nó que tem o objeto que temos que retornar é agora a cabeça, tornando assim ‘imutavel’, isto é, mais nenhuma thread pode tentar adquirir o objeto deste nó porque este é a head. Depois do nó ser a cabeça metemos o seu value a null, devido ao desreferenciamento deste objeto o garbage collector pode desalocar o mesmo se não houver mais referencias para este, e retornamos o antigo valor.

boolean isEmpty() – o método retorna true se o campo next do head for null.

O teste usado foi o fornecido nos anexos. Este programa foi feito em java e c# e ambos têm a mesma lógica

**Exercício 2**

Na implementação deste exercício, foi usado um AtomicInteger para controlar o número de threads que estavam a usar aquela variável. É usado uma técnica em que primeiro se lê a variável, de sequida toma-se ações baseadas nessa observação. Se uma dessas ações for alterar o valor do atomicInteger é usado o compareAndSet, de modo a ser thread safe. Este exercício foi feito em Java e encontra-se no package exercicio2.

**Exercicio 3**

Este exercício encontra-se dentro da solução ConcurrentQueue, no projeto ExpirableLazy. Nesta implementação foi usado um memory barrier entre o incremento dos waiters e a observação do objecto pair que contem o value que ia ser retornado. Este memory barrrier foi aqui colocado de modo a não haver troca de ordem entre estas duas instruções. Esta ordem é importante porque quando o valor é calculado é atualizado primeiro o value, e só depois iremos verificar os waiters. Como estes tem ordens inversas não haverá bloqueios apesar do value já ter sido calculado. Na segunda ordem de instruções não foi necessário o uso do memoryBarrier porque devido as dependências de instruções não era possível a sua reordenação. Esta implementação é semi-free-lock, isto é, nos casos otimistas, em que ninguém está a calcular o valor ou o valor se encontra válido, não é necessário entrar no lock.

**Exercicio 4**

Este exercício foi feito em java e encontra-se no package exercicio4. O teste usado foi o fornecido nos anexos.