Universidade Federal do Rio Grande do Norte - UFRN
Instituto Metrópole Digital - IMD
Bacharelado em Tecnologia da Informação - BTI
Programação Concorrente - DIM0612

Joaliton Luan Pereira Ferreira

Relatório do Projeto II de Programação Concorrente
Controle de Acesso de Dados

Docente: Everton Ranielly de Sousa Cavalcante

Natal,
Maio de 2017

1. Introdução

Este relatório tem como objetivo explicitar a metodologia utilizada para o desenvolvimento do projeto; as pesquisas realizadas, materiais utilizados como bases a serem seguidas e o aprendizado adquirido no decorrer da segunda unidade da disciplina. Bem como os detalhes de implementação; da organização dos arquivos e do funcionamento do programa; e os resultados e conclusões obtidos. Visa ainda apresentar as dificuldades e problemas enfrentados no desenvolvimento, assim como suas respectivas soluções.

2. Detalhes de implementação

O projeto foi implementado seguindo boas práticas de programação, utilizando-se da linguagem Java e suas bibliotecas. A implementação foi totalmente baseada em modularização de arquivos, a coesão e o acoplamento das classes e métodos. Dois pacotes foram criados para conter cada uma das soluções dos problemas, são eles: *unisexToiletProblem* e *concurrentLinkedList*.

2.1 Detalhes gerais de implementação - Problema do banheiro Unissex

Este pacote contém todas as classes utilizadas para resolver o problema concorrente conhecido como "problema do banheiro unissex". Para esta solução foram implementadas as classes que representam o banheiro (*Toilet)*, uma pessoa (*Person)* e o gênero da pessoa (*Gender*). A seguir serão detalhadas cada classes:

- *Toilet:* Uma classe que define os atributos e métodos utilizados para representar um banheiro no problema. Ela é a responsável por escalonar e controlar as pessoas na fila, além de ser responsável por implementar um meio de combater possíveis casos de inanição (*starvation*) que poderiam ocorrer.
- *Person*: Uma classe que define os atributos e métodos utilizados para representar uma pessoa que deseja utilizar o banheiro. Esta classe é responsável por, basicamente, definir o tempo que uma pessoa leva ao usar o banheiro.
- *Gender:* Uma classe do tipo *enum* que define os gêneros utilizados no problema. Esta classe contém alguns métodos que facilitam o uso de gêneros aleatórios.

2.2 Detalhes do uso de controles de concorrência - Problema do banheiro Unissex

Para resolver este problema, foram utilizados estruturas controladoras de concorrência do tipo semáforo, de modo que há um semáforo que proporciona aquisição FIFO e permite até uma quantidade *X* de pessoas do mesmo sexo dentro do banheiro, que ficam no banheiro por um tempo aleatório. Há também um contador *C* que previne que hajam casos de *starvation* sendo decrementado a cada vez que uma pessoa do mesmo sexo entra no banheiro. Ao ser zerado, o contador impede que novas pessoas entrem no banheiro até que este esteja vazio. Após o banheiro esvaziar, a execução

normaliza o contador e prossegue com o próximo da fila, repetindo o passo anterior quando o contador zerar novamente.

2.3 Detalhes gerais de implementação - Lista ligada concorrente

Este pacote contém todas as classes utilizadas para resolver o problema concorrente da "lista ligada concorrente". Para esta solução foi implementada a classe que representa a lista ligada concorrente (*ConcurrentLinkedList*). A classe será detalhada a seguir:

- *ConcurrentLinkedList:* Uma classe que define os atributos e métodos utilizados para representar uma lista ligada concorrente. Ela é a responsável por controlar os acessos simultâneos que podem ocorrer na lista ligada durante uma busca, inserção e/ou remoção.

2.2 Detalhes do uso de controles de concorrência - Lista Ligada Concorrente

Para resolver este problema, foram utilizados estruturas controladoras de concorrência do tipo bloqueios explícitos e variáveis de condição. Existem bloqueios explícitos para remoção e para inserção. Variáveis de condição foram utilizadas para controlar e sincronizar as regiões críticas que envolvem a busca e a remoção. Explicando como cada um dos três métodos funciona:

- Remoção: A remoção sempre ocorre em uma região sincronizada (*synchronized*) de acordo com uma variável de condição *condToRemove* que torna a remoção *thread-safe* em relação a busca e em relação à variável booleana *couldSearch* que indica se novas buscas podem ocorrer ou não (as buscas ficam esperando caso *couldSearch* as impeça de buscar). Além disso, há bloqueios explícitos que impedem que novas remoções sejam iniciadas e que novas inserções sejam iniciadas até o fim da remoção em execução. Ao final da remoção cada bloqueio (inserção e remoção) é liberado e cada thread de busca é notificada.
- Inserção: Para a inserção, há um bloqueio explicítio de inserção que impede que novas inserções iniciem até o fim da inserção em execução.
- Busca: A busca ocorre sempre em uma região sincronizada (*synchronized*) de acordo com uma variável de condição *condToRemove* que torna a busca *thread-safe* em relação a remoção e em relação à variável booleana *couldSearch* que indica se uma nova busca pode iniciar ou não. Caso uma nova busca não posso iniciar, ela entra em espera (*wait*) até que a thread responsável pela remoção notifique, liberando todas as threads de busca.

3. Instruções de execução

O projeto foi implementado utilizando a IDE eclipse. Para executar cada uma das soluções, executar as classes *Main.java* presentes em cada pacote.

4. Considerações finais

Este projeto teve como intuito pôr em prática as técnicas, ferramentas e conhecimentos adquiridos pelo autor sobre controle e acesso de dados concorrentes ao longo da unidade 2 da disciplina de programação concorrente. As implementações aqui descritas estão em acordo com o proposto na atividade e condizem com o bom uso de técnicas de controle de acesso a dados concorrentes. Espera-se que em todos os casos de execução para ambas as soluções apresentadas nos pacotes, os objetivos almejados sejam alcançados sem problemas.