Sincronização em Programas Concorrentes

DIM0612 - Programação Concorrente

Raquel Lopes de Oliveira

Novembro de 2017

Introdução

Este trabalho consistiu na elaboração de algoritmos e suas implementações para dois programas concorrentes, utilizando diferentes métodos de sincronização.

A linguagem utilizada para implementação foi Java, pois dispõe de mecanismos nativos para sincronização que facilitam a implementação.

Solução

Foi criado uma classe *Person* que possui uma identificação para o sexo (restringindo para 'Men' e 'Women' para representar o sexo masculino e feminino). Essa classe implementa *Runnable* e na sobrecarga do método *run()* o sleep é chamado para representar o tempo que essa pessoa passa no banheiro; após, é chamado o método do banheiro responsável por permitir a saída da pessoa do banheiro.

Uma classe abstrata *Bathroom* foi criada com as implementações em comum da representação do banheiro. Inicialmente foi pensando em ter um atributo no banheiro para representar se ele estava sendo ocupado por pessoas do SEX = Male ou SEX = Women, mas por uma questão de debug e preferencia pessoal foi criado dois atributos, num_men e num_women para representar quantos homens e quantas mulheres estariam dentro do banheiro. Foi optado por usar uma ConcurrentLinkedQueue para a lista de espera para preservar a ideia de 'first-in and firs-out' e uma List para as pessoas que estão dentro do banheiro, pois uma pessoa que entrou depois poderia sair antes de outra que entrou antes.

Duas soluções foram criadas, uma com o mecanismo de sincronização semáforo e outra através do monitor.

Semáforo

Lógica

Nessa solução, foi criado um semáforo para o banheiro. A capacidade do semáforo é 5, mas pode ser modificada no arquivo *MainSemaphore.java*. Para realizar o experimento 3*5(3 é uma constante,multiply, definida em Bathroom e 5 a capacidade setada em MainSemaphore.java) pessoas são criadas aleatoriamente tanto em respeito ao sexo quando ao tempo de duração no banheiro(0.1 a 5.0) através do método *populate()* em Bathroom.java e do construtor da classe Pessoa.
Talvez a chamada do populate não precisasse estar no método *run()* da classe *BathroomSemaphore*, mas sim no seu construtor, por exemplo. Mas a função principal desse método é de fazer com que todas as pessoas na lista de espera possam usar o banheiro, respeitando a ordem de entrada (não pode "furar fila") e sem permitir que pessoas de sexo distinto usem o banheiro simultaneamente.

Dificuldades

Por falta de atenção fiquei um tempo sem entender o porquê de não estar funcionando e dado a análise do contexto (observando os valores de num_men, num_women e demais atributos) notei que no método *run()*

da classe BathroomSemaphore eu estava retirando as pessoas da lista de espera, mesmo que na chamada do método *receive(Person p)* a pessoa não tivesse entrado de fato do banheiro. Mas isso foi facilmente solucionando apenas mudando o retorno da função para confirmar se a pessoa em questão estava apta para entrar no banheiro e só então ser retirada da lista de espera.

Corretude

Para garantir que o problema esteja correto é necessário que o programa satisfaça 4 propriedades: utilização de um mecanismo de exclusão mutua; ausência de *deadlock*, ausência de *starvation* e a impossibilidade de fazer suposição sobre a velocidade de execução dos processos.

O semáforo é responsável por controlar quem entra e quem sai do banheiro. Ele garante que apenas o número definido na capacidade(5) seja capaz de ocupar o banheiro ao mesmo tempo.

Não há *starvation* pois o semáforo do banheiro é com justiça; é implementada uma lógica FIFO para que as pessoas entrem na lista de espera e consequentemente possam usar o banheiro na ordem em que foi pedido/requisitado(nesse caso na ordem de criação das pessoas); além disso cada pessoa tem um tempo determinado para usar o banheiro. Não há como alguém ficar esperando indeterminadamente para usar o banheiro.

Através do limite de pessoas no banheiro e o fato de não permitir que se possa furar fila é possivel garantir que não haja *deadlock* pois nenhuma thread ficará com um recurso impedindo outras de usar enquanto só outra poderia fazer ela liberar o recurso.

Monitor

Lógica e Corretude

A lógica do monitor foi a mesma para da usada com os semáforos, mas as variavéis que sofrem mudança durante o processo foram modificadas para volateis e os métodos usados em *BathroomMonitor* possuem agora na sua com assinatura o termo syncronized que permitem a prevenção de erros de consistência e interferencia de thread.

Análise Comparativa

Apesar da mudança, na implementação, ser pequena entre as duas abordagens. Eu demorei a entender a ideia do monitor e de como ter certeza que de fato a implementação estava correta. A ideia de semáforos me parece muito mais intuitiva e fácil.
Inicialmente pensei em uma das abordagens ser o bloqueio explicíto, mas apesar do problema não ser tão difícil a ideia de deixar ao cargo do programador verificar se não está deixando um caso de uso de lado/esquecido aparenta ser uma escolha mais perigosa. Quando existe um mecanismo que já previne certo problemas, o programador (normalmente) tende a escolher essa abordagem.

Compilação e execução

Foi usado o Maven para a configuração dos packages e bibliotecas. É necessário do compilador *javac* e do Java 9. Você pode abrir o projeto em sua IDE de preferência importando o projeto através do Maven. Ou pode executar as seguintes instruções em linha de comando:

```
javac -d build/classes -cp build/classes src/main/java/*.java src/util/java/*.java -Xlint
```

Para a implementação com semáforo:

```
java -cp build/classes/  MainSemaphore
```

Para a implementação com monitor:

```
java -cp build/classes/  MainMonitor
```