UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
BACHARELADO EM ENGENHARIA DE SOFTWARE

Trabalho prático

Mecanismos de Sincronização

Rita de Cássia Lino Lopes

Natal - RN
Dezembro/2022

Sumário

1. Introdução

Este relatório refere-se ao projeto e desenvolvimento de uma solução para um problema relacionado a banheiro unissex. Tal problema foi proposto como atividade avaliativa da disciplina de Programação Concorrente e visa mensurar os conhecimentos adquiridos pelos alunos sobre mecanismos de sincronização de processos e threads.

Ainda nesta seção será descrito brevemente o problema e na posterior será apresentado o Desenvolvimento, explicando detalhes da implementação e escolhas do programador para garantir a sincronização. Em seguida serão apresentados os Resultados, nos quais explicitamos as formas de compilação e execução do projeto juntamente com a garantia de corretude e algumas dificuldades encontradas. Por fim, este relatório apresentará as considerações finais.

O problema do banheiro unissex trata-se da necessidade de utilização de um banheiro por ambos os sexos: feminino e masculino, de forma que não exceda a capacidade máxima do banheiro e, principalmente, não aconteça de ambos sexos estarem simultaneamente utilizando o banheiro. Assim, a proposta é a utilização de concorrência para simular tal situação e garantir que as condições sejam cumpridas eficientemente.

Dessa forma, a solução proposta encontra-se no repositório do Github que pode ser acessado no link https://github.com/ritallopes/pconcorrente_t3 e mais detalhes da implementação serão vistos a seguir neste trabalho.

2. Desenvolvimento

Nesta seção, será descrito o processo de implementação e explicadas algumas motivações para as escolhas realizadas. Inicialmente, com a seção de Implementação para mais livremente explanar o desenvolvimento da solução e, em seguida, Sincronização para explicar o mecanismo escolhido.

Implementação

Inicialmente, a linguagem de programação foi escolhida dentre as opções C++, Python e Java, o desenvolvedor tinha muito pouca experiência com uso de Threads em C++, já em Python justamente as experiências anteriores contribuíram para a busca de uma nova opção. Dessa forma, a implementação foi feita em Java. A decisão por esta linguagem de programação foi influenciada pela maior facilidade de lembrar alguns conceitos, já que foi a

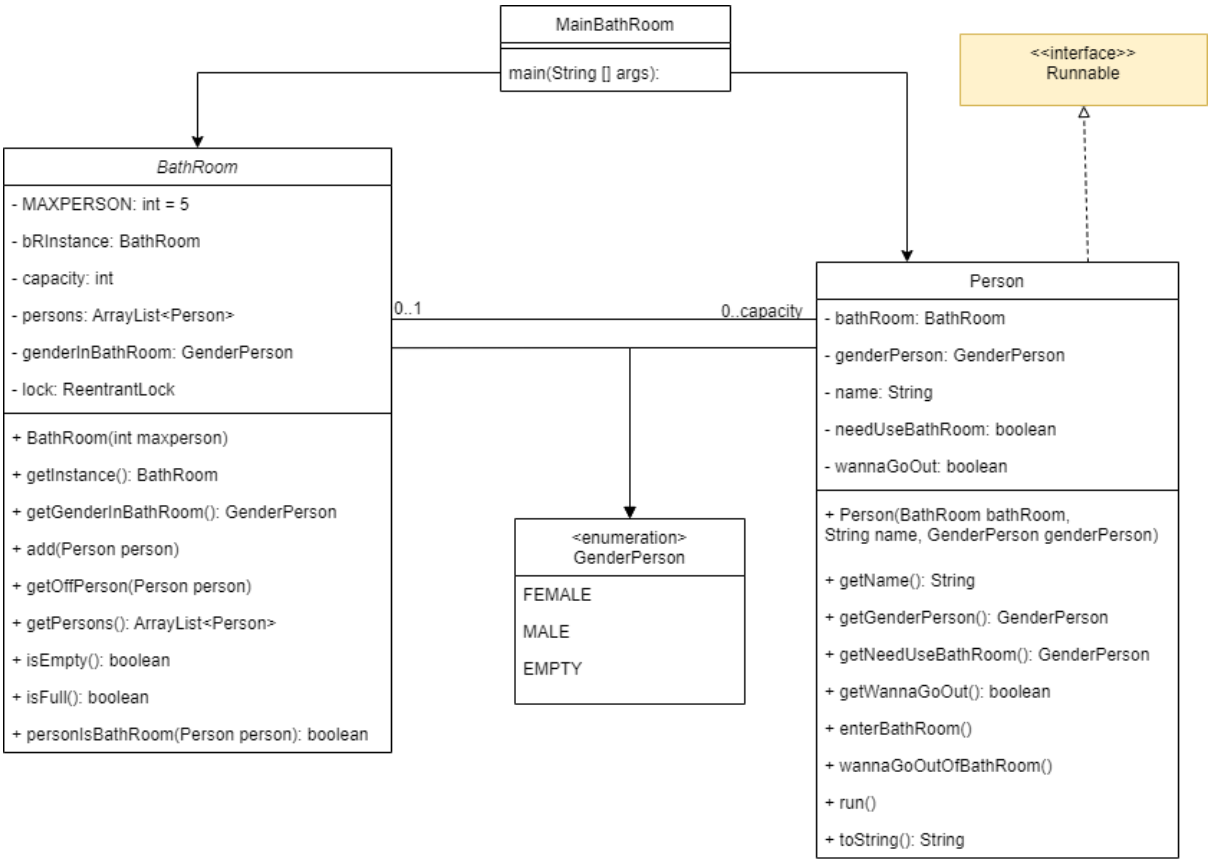
usada em exemplos mais recentes apresentados em sala de aula mais e, costumeiramente, utilizada em outras disciplinas do curso.

O ambiente de desenvolvimento utilizado foi a IDE Intellij IDEA da JetBrains e a mesma foi utilizada para auxiliar na geração da documentação do projeto. Essa documentação foi feita a partir do código fonte por meio do Javadoc. No repositório, pode ser acessada na pasta *docs*.

É importante citar um ponto: que detalhes da escolha do mecanismo de sincronização são mostrados na seção seguinte, mas nesta ainda será necessário adiantar alguns pontos para explicar a interação entre componentes do programa. Dito isso, seguimos com explicações da organização do projeto.

Os componentes principais do projeto, mostrados na Figura 1, que representa o diagrama de classes do projeto, são duas classes relacionadas com o domínio do problema: a *BathRoom*, representando o banheiro e a *Person*, representando as pessoas que usam o banheiro. Além disso, temos uma enumeração chamada *GenderPerson* representando o sexo das pessoas, ela foi feita separada para favorecer a clareza e o reaproveitamento de código. Por fim, a *MainBathRoom* é a classe com o método estático *main*.

Figura 1: Diagrama de classes da solução



Fonte: Autoral

A lógica da interação entre os elementos citados anteriormente acontecem da seguinte forma: em *MainBathRoom* há leitura da capacidade máxima do banheiro, caso seja informada como argumento para o programa (caso não seja informado está predefinido como 5 pessoas no banheiro simultaneamente). Em seguida, há a instanciação de um objeto da classe *BathRoom* passando a capacidade como parâmetro para um construtor parametrizado. Ainda no método *main* são criados objetos do tipo *Person*, alguns do sexo masculino (quantidade definida pelo segundo argumento passado para o programa, caso ausente o número de homens será 5), outros do sexo feminino (semelhante ao masculino, quantidade definida pelo terceiro argumento). Como parâmetro para o construtor desses objetos da classe Person são passados o mesmo objeto da classe *BathRoom*, um nome fictício da pessoa e um dos sexos da enumeração GenderPerson.

Ademais, todos os objetos do tipo *Person* são individualmente adicionados numa lista e posteriormente são criadas Threads, uma para cada objeto da lista. Como *task* é passado para cada Thread um objeto *Person*, essa classe implementa a interface *Runnable* e, então, podemos considerar cada "Pessoa" como uma Thread. Para finalizar o método *main*, todas as Threads têm seus métodos *start* acionados e em seguida o *join.*

Por sua vez, na classe *Person*, o método *run()* fazendo a chamada do método que tenta entrar no banheiro, por meio da chamada do método *add()* do objeto do tipo *BathRoom*, que contém o mecanismo de sincronização *Lock*. Esse mesmo mecanismo é utilizado após um tempo randômico entre 0,5 e 10 segundos (limite de tempo que uma pessoa permanecer dentro do banheiro), pelo método *getOffPerson()* para remover a pessoa da lista de pessoas no banheiro. Para garantir que quando o banheiro estiver sendo usado por um sexo apenas pessoas do mesmo sexo poderão entrar, o objeto do tipo *BathRoom* possui um atributo do tipo *GenderPerson*, cujo valor é referente ao sexo das pessoas no banheiro em dado momento. Além disso, antes de cada pessoa entrar no banheiro é verificado se ele está em sua capacidade máxima.

Por fim, quando todas pessoas de um mesmo sexo saem do banheiro, as Threads com pessoas do outro sexo podem executar o bloco de código que as insere na lista de pessoas que estão no banheiro.

Sincronização

Apesar de ter sido cogitado e brevemente analisado o uso de *Semaphores*, *CyclicBarrier* e *Locks ,* o desenvolvimento foi iniciado pensando na utilização de *Monitors*. No entanto, ao precisar decidir o local de inserção da palavra reservada *synchronized* para

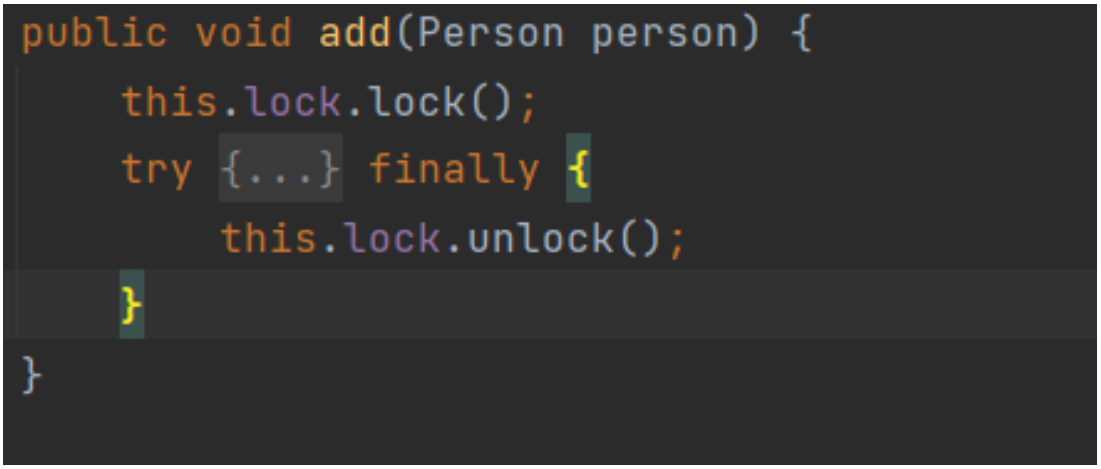
isolar a região crítica e garantir a exclusão mútua, foram encontradas algumas dificuldades e, por consequência, começou-se a pesquisar as possibilidade com tal mecanismo. Por fim, ao entender mais claramente o problema à medida que estavam sendo testadas formas de garantir as exigências, os mecanismos foram revistos e a implementação foi feita utilizando o *Lock*.

A utilização do *Lock*, do pacote padrão do Java java.util.concurrent.locks.Lock, trouxe algumas vantagens para a solução do problema quando comparado ao uso do *synchronized*. Pois possibilitou maior flexibilidade ao planejar os métodos utilizados para execução das ações do problema, isso porque não era necessário concentrar todo o bloco considerado região crítica em um só método. Ademais, acredita-se que um dos aspectos mais importantes para optar pelo *Lock* é a possível garantia de justiça.

A localização do bloqueio, assim como, o momento do desbloqueio é um dos aspectos mais importantes para garantir a eficiência da solução. Primeiramente, foi necessário analisar o problema para perceber as ações que poderiam disputar recursos com outra ação. Assim, como resultado anotamos os blocos que envolvem a adição e remoção de pessoas no banheiro como regiões críticas.

A primeira ação, uma pessoa entrar no banheiro, necessitava de exclusão mútua para que não fosse possível a entrada de duas pessoas no banheiro ao mesmo tempo, ou seja, não haver a adição de mais de uma *Thread* no atributo *persons* ao mesmo tempo. Pois, caso isso não ocorresse, quando o banheiro estivesse vazio, poderia entrar um homem e uma mulher ao mesmo tempo e, consequentemente, haver pessoas de dois sexos simultaneamente no banheiro. Além disso, a ausência de *lock* nesse método poderia permitir a entrada de uma quantidade maior de pessoas do que a capacidade do banheiro. Dessa forma, como mostrado na Figura 2, quando o método é iniciado ocorre a chamada do *lock* e, independentemente de ter ocorrido algum erro, há a chamada do *unlock* assim que o bloco de adição de pessoas é executado.

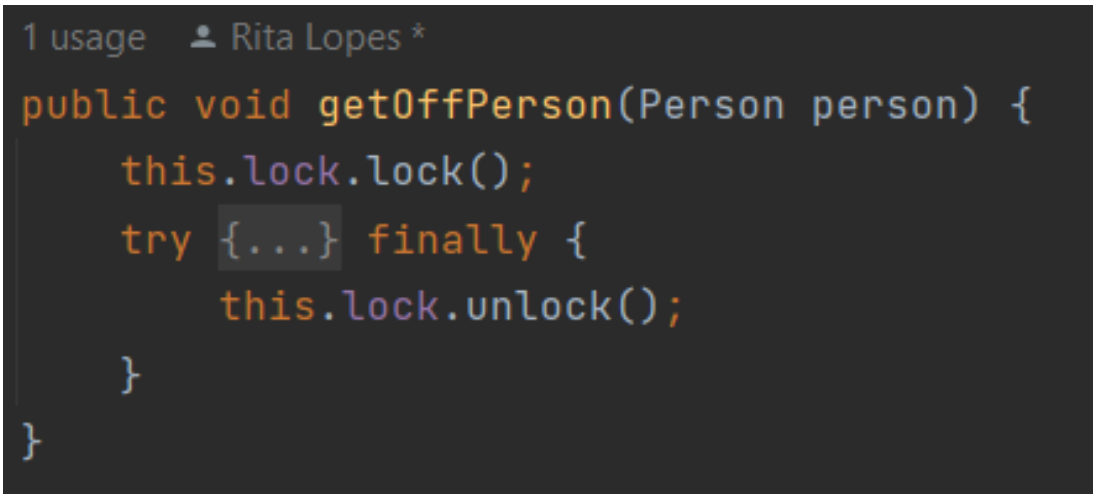
Figura 2: Utilização do Lock na adição de pessoa no banheiro



Fonte: Autoral

Por sua vez, a saída de pessoas do banheiro também possui blocos que podem resultar em conflitos de compartilhamento de recursos. Portanto, também é necessária a utilização de bloqueio nessa parte, mais especificamente, na parte de remoção de *Thread* no conjunto de pessoas, atributo *persons*. Como podemos observar na Figura 3, há um *lock* na primeira linha do método e um *unlock* em um bloco *finally* para garantir a destrava mesmo que haja alguma exceção na remoção que acontece dentro do bloco *try*, que está oculto na figura.

Figura 3: Utilização do Lock na remoção de pessoa no banheiro



Fonte: Autoral

3. Resultados

Garantia de corretude

Como já explicado na seção anterior, a classe *Person* implementando a interface Runnable e o uso de uma Thread por objeto desta classe garante a execução concorrente da solução. No entanto, isso não bastaria para garantir a eficácia da solução, por isso, a utilização do mecanismo de sincronização *Lock* nas áreas críticas, mais especificamente com a classe *ReentrantLock*, nos traz a garantia de justiça e exclusão mútua. Além disso, para evitar *deadlocks* as instâncias da classe *Person* não interagem entre si e mesmo que ocorra algum erro que lance exceção o bloco da região crítica é sempre liberado.

Como ilustração corretude da solução, nas imagens abaixo, Figura 4, Figura 5 e Figura 6, mostramos exemplos da execução com diferentes argumentos passados para o programa. Se observamos mais atentamente será possível notar que o primeiro a entrar nem sempre é o primeiro a sair devido ao tempo randômico, a exemplo Figura 4, pessoa de nome "H0", do sexo masculino é o primeiro a entrar e o último a sair, e somente após sua saída a primeira pessoa do sexo feminino entra("M2").

Figura 4: Execução banheiro com capacidade para 2, 5 homens e 5 mulheres

```
H0 entrou no banheiro
1 pessoas no banheiro.
H3 entrou no banheiro
2 pessoas no banheiro.
H4 entrou no banheiro
3 pessoas no banheiro.
H1 entrou no banheiro
4 pessoas no banheiro.
H2 entrou no banheiro
5 pessoas no banheiro.
Agora o banheiro encheu
H3 usou o banheiro
H4 usou o banheiro
H2 usou o banheiro
H1 usou o banheiro
H3 saiu!
H4 saiu!
H2 saiu!
H0 usou o banheiro
H1 saiu!
H0 saiu!
O Banheiro está vazio
M2 entrou no banheiro
```

Fonte: Autoral

Na Figura 5, por sua vez, podemos observar detalhes da saída do programa, informando quando cada pessoa entra no banheiro e quantas pessoas há no interior do banheiro. Além disso, também é informado quando o banheiro atinge sua capacidade máxima e quando ele está vazio, possibilitando a entrada de pessoas de outro sexo.

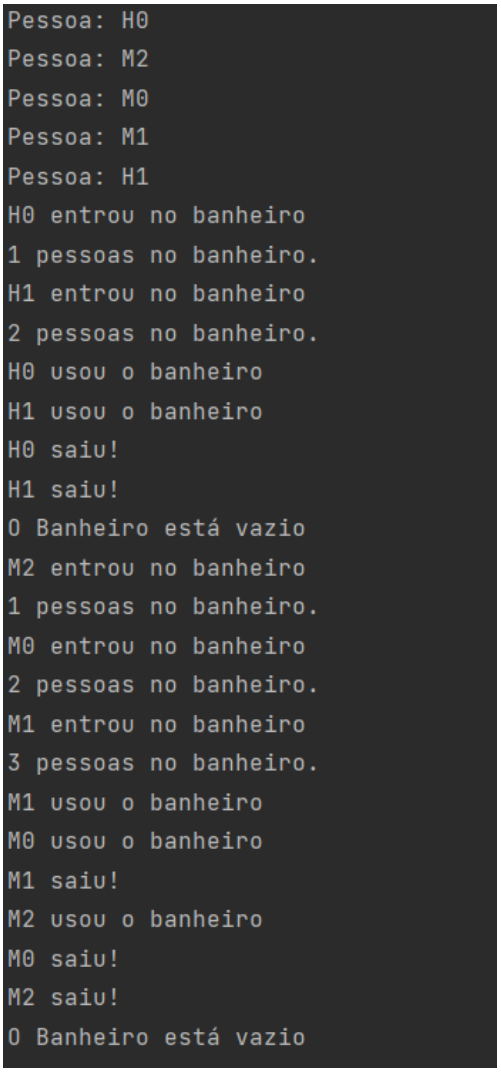
Figura 5: Execução banheiro com capacidade para 2, 2 homens e 3 mulheres

```
Pessoa: M1
Pessoa: H0
Pessoa: M2
Pessoa: H1
Pessoa: M0
H0 entrou no banheiro
1 pessoas no banheiro.
H1 entrou no banheiro
2 pessoas no banheiro.
Agora o banheiro encheu
H1 usou o banheiro
H0 usou o banheiro
H1 saiu!
H0 saiu!
O Banheiro está vazio
M0 entrou no banheiro
1 pessoas no banheiro.
M2 entrou no banheiro
2 pessoas no banheiro.
Agora o banheiro encheu
M2 usou o banheiro
M0 usou o banheiro
M2 saiu!
M1 entrou no banheiro
2 pessoas no banheiro.
Agora o banheiro encheu
M0 saiu!
M1 usou o banheiro
M1 saiu!
O Banheiro está vazio
```

Fonte: Autoral

Por fim, na Figura 6, é a simulação de uma situação em que a quantidade de pessoas para usar o banheiro é inferior à sua capacidade. No exemplo, a capacidade máxima definida é 5, mas somente 2 homens e 3 mulheres solicitam a entrada. No entanto, mesmo em tal situação, primeiramente entrará pessoas de um sexo e posteriormente a de outro.

Figura 6: Execução banheiro com capacidade para 5, 2 homens e 3 mulheres



Fonte: Autoral

Dificuldades durante o desenvolvimento

A dificuldade inicial foi optar entre os mecanismos de sincronização, especialmente o *Monitor* e o *Lock*. Para isso, foi necessário revisar as características de cada um e entender cada ponto e necessidade do problema. Somente após isso, o *Lock* foi visto como melhor opção. No entanto, ainda há desejo por parte do desenvolvedor de buscar outro esquema para

utilização do *Monitor*, especialmente, para entender, numa prática mais específica, as diferenças entre eles.

Ademais, outra dificuldade encontrada, mas já no momento da implementação foi identificar em qual parte do código colocar o tempo randômico para cada pessoa permanecer no banheiro. No fim, foi colocado na classe *Person,* no método *enterBathRoom()*, após a impressão da entrada da pessoa e antes da impressão do "usou o banheiro". Dessa forma, fora do bloco de bloqueio.

Compilação e execução

As classes para compilação estão dentro da pasta *src/lock* do projeto. Podem ser vistas no repositório por meio do link https://github.com/ritallopes/pconcorrente_t3/tree/main/src/lock . Para compilar pode ser utilizada uma IDE que facilite o processo ou o *javac,* como ilustra a Figura 7.

Figura 7: Exemplo de compilação

```
\BES\pconcorrente\src\lock>javac MainBathRoom.java GenderPerson.java BathRoom.java Person.java
```

Fonte: Autoral

Para execução basta utilizar o comando `java -jar pconconrrente.jar <capacidadeDoBanheiro> <quantidadeSexoMasculino> <quantidadeSexoFeminino>`, no qual pconcorrente.jar é o executável disponível no repositório, https://github.com/ritallopes/pconcorrente_t3/blob/main/pconcorrente.jar, e os argumentos são, respectivamente, a capacidade máxima de pessoas simultaneamente no banheiro, a quantidade de pessoas do sexo masculino e a quantidade de pessoas do sexo feminino. Na Figura 8, temos o exemplo da execução com o comando `java -jar pconcorrente.jar 3 5 3`, nele podemos observar o início da saída do programa.

Figura 8: Exemplo de execução

```
BES\pconcorrente>java -jar pconcorrente.jar 3 5 3
Pessoa: H4
Pessoa: H1
Pessoa: H2
Pessoa: H0
Pessoa: H3
Pessoa: M2
Pessoa: M0
Pessoa: M1
```

Fonte: Autoral

4. Considerações finais

Destarte, apesar das dificuldades enfrentadas a solução foi implementada seguindo os requisitos exigidos na descrição. Ainda, buscou-se utilizar o melhor mecanismo de sincronização de Threads para o problema descrito a fim de garantir a corretude do programa e a ausência de erros e exceções não tratadas.

O projeto disponível no repositório do github, que pode ser acessado por meio do link https://github.com/ritallopes/pconcorrente_t3, contém todo o código fonte implementado além da documentação.