

UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE RIO PARANAÍBA
SISTEMAS DE INFORMAÇÃO

2746 - RODRIGO DE OLIVEIRA COSTA
5152 - LARISSA SOARES BONFIM
6010 - JOÃO VITOR ARIEIRA
6011 - THIAGO MOREIRA DE SOUZA

TRABALHO PRÁTICO 2

RIO PARANAÍBA
2019

2746 - RODRIGO DE OLIVEIRA COSTA

5152 - LARISSA SOARES BONFIM

6010 - JOÃO VITOR ARIEIRA

6011 - THIAGO MOREIRA DE SOUZA

TRABALHO PRÁTICO 2

Trabalho Prático 2 apresentado à Universidade Federal de Viçosa como parte das exigências para a aprovação na disciplina SIN142 – Programação Concorrente e Distribuída

Orientador: Prof. João Batista

RIO PARANAÍBA

2019

Lista de ilustrações

Figura 1 – Como importar um projeto no NetBeans.	6
Figura 2 – Resultado do programa executado sequencialmente.	8
Figura 3 – Resultado do programa executado sem utilização de semáforo.	9
Figura 4 – Resultado do programa executado com utilização de semáforo.	9

Sumário

1	Introdução	4
2	Referencial Teórico	5
2.0.1	Programação Concorrente em Java	5
3	Desenvolvimento	6
3.0.1	Manual de utilização	6
3.0.2	Explicação do código-fonte	7
4	Resultados e Conclusão	8
Referências						10

1 Introdução

A medida que a tecnologia avança, novas aplicações surgem. Algumas aplicações são de uso mundial, como as redes sociais. Elas, por sua vez, têm a capacidade de compartilhar uma enorme quantidade de informações diariamente, como comentários, conquistas e críticas que devem ser atualizadas sem muita demora (BURCKHARDT et al., 2012). Scott (2017) em *Why Computers Can't Count Sometimes* convida a olhar os números que ficam nas redes sociais, especificamente os marcadores de popularidade, como *likes*, gostei, número de comentários, etc. e entender melhor como o computador os contabiliza, o porque deles aumentarem consideravelmente e podendo diminuir, e também o porque da diferença destes números em dispositivos diferentes. O autor desmitifica o fato do computador ter o propósito de funcionar como uma calculadora, levando a crer que a contagem deveria ser uma atividade fácil. Porém, segundo Scott, quando trata-se de escala, quando solicitações de contagens aumentam, esta tarefa se torna bastante complicada.

No vídeo, Tom, a respeito do *YouTube*, ao receber dois pedidos de "Gostei" ao mesmo tempo, são atribuídos dois segmentos, um pra cada solicitação. Ele explica que um segmento pode estar lendo o valor do contador, por exemplo, 68, enquanto o outro já leu e está atualizando, 68+1, dessa maneira o mesmo número, 69, voltaria a ser escrito no banco de dados o 69 e não 70, perdendo assim um "Gostei". Em geral quando um vídeo possuir muita popularidade aconteceria muitas colisões como estas (SCOTT, 2017). Um requisito muito difundido na computação é que uma aplicação mantenha atualizações o tempo todo destas informações. Porém, infelizmente as conexões dos servidores são propensas a lentidão e indisponibilidade, assim este requisito muitas vezes não é cumprido (BURCKHARDT et al., 2012).

Para solucionar o problema de colisões ele propõe uma fila de espera, mas para um sistema como o *YouTube* poderia não comportar tamanha espera dos usuários para assistirem os vídeos. Então ele propõe uma solução mais adequada chamada *eventual consistency*, no qual muitos servidores, ao invés de relatar todas as visualizações de imediato, cada servidor vai ter sua própria contagem e assim que possível atualizaria o servidor principal. Ou seja, para garantir a capacidade de resposta do acesso de dados compartilhados do servidor, muitos aplicativos mantêm réplicas locais dos dados que permanecem instantaneamente acessíveis independente da lentidão ou indisponibilidade.

Apesar de sua aparente simplicidade, esse cenário de uso e atualização de dados pode ser surpreendentemente desafiador (BURCKHARDT et al., 2012). Frente ao exposto, este projeto de pesquisa visa discutir um mecanismo que assegura a execução de requisições ao contador de *like* de modo que a consistência de dados seja mantida utilizando a linguagem de programação java e a técnica de semáforo.

2 Referencial Teórico

2.0.1 Programação Concorrente em Java

Além do livro-texto da disciplina explicar a parte teórica, ele também mostra a aplicação dos conceitos apresentados em várias linguagens de programação. Através dos exemplos de cada linguagem, optou-se por utilizar a linguagem de programação Java. Foram criadas extensões da classe *Thread* e instância da classe *Semaphore*.

Um objeto do tipo *Thread* pode ser inicializado durante a execução de um programa. Este objeto, como o próprio nome já diz, é uma *thread* de execução neste programa. A Java Virtual Machine (JVM) permite que um programa possua múltiplas *threads* sendo executadas de forma concorrente ([ORACLE, 2019](#)). Não iremos nos estender neste tópico, pois o mesmo já foi abordado no projeto 1 desta disciplina.

Teoricamente ([BEN, 2006](#)) um semáforo S serve para controlar as entradas e saídas na seção crítica de cada *thread*, possuindo dois atributos principais: S.V referente a um recurso disponível para uma *thread* consumir em algum momento de seu pré-protocolo e S.L referente à uma lista de processos que tentaram consumir um recurso indisponível (S.V = 0). Um semáforo possui dois métodos principais: S.wait() tenta consumir um recurso, caso ele esteja disponível, permite um processo entrar em sua seção crítica (decrementando uma unidade em S.V), caso contrário o processo tem seu estado alterado para *blocked* e o mesmo é adicionado à S.L; S.signal() incrementa uma unidade em S.V e remove ou arbitrariamente um elemento de S.L (semáforo fraco) ou o primeiro elemento de S.L (semáforo forte).

Na prática é um pouco diferente. Em Java o desenvolvedor decide se um semáforo é forte ou fraco através do construtor da classe *Semaphore*, podendo ou não passar como parâmetro um valor booleano *fair*. Este por padrão é inicializado como *false*. Caso *fair* seja *true* o semáforo criado será um semáforo forte, caso contrário será um semáforo do tipo *busy-wait* (este tipo de semáforo não possui o atributo S.L mencionado no parágrafo anterior). Os métodos S.wait() e S.signal() apresentados anteriormente são implementados na classe *Semaphore* pelos métodos *acquire()* e *release()*, respectivamente. Caso o método *acquire()* seja chamado antes de uma *thread* ser executada e um recurso não esteja disponível, a *thread* será interrompida.

3 Desenvolvimento

3.0.1 Manual de utilização

Para execução dos códigos vai ser necessário a instalação na maquina o *NetBeans IDE 8.2* e que os arquivos *atualizarValores.txt* e *diminuirValores.txt* estejam na pasta raiz do projeto. Nesse caso os arquivos *atualizarValores.txt*, *diminuirValores.txt* e o algoritmo *sin142trabalhoPratico2GeraValores.c* já estão anexados na raiz do projeto. O trabalho pode ser encontrado para download no repositorio a seguir: <<https://github.com/rodrigo-UFV-2746/ProjetoConcorrente-2-SIN142>>.

Passos para abrir o projeto: Abra o *NetBeans*, vá em Arquivo, Importar Projeto, De ZIP, Selecione o arquivo e clique em Importar, conforme a figura 1.

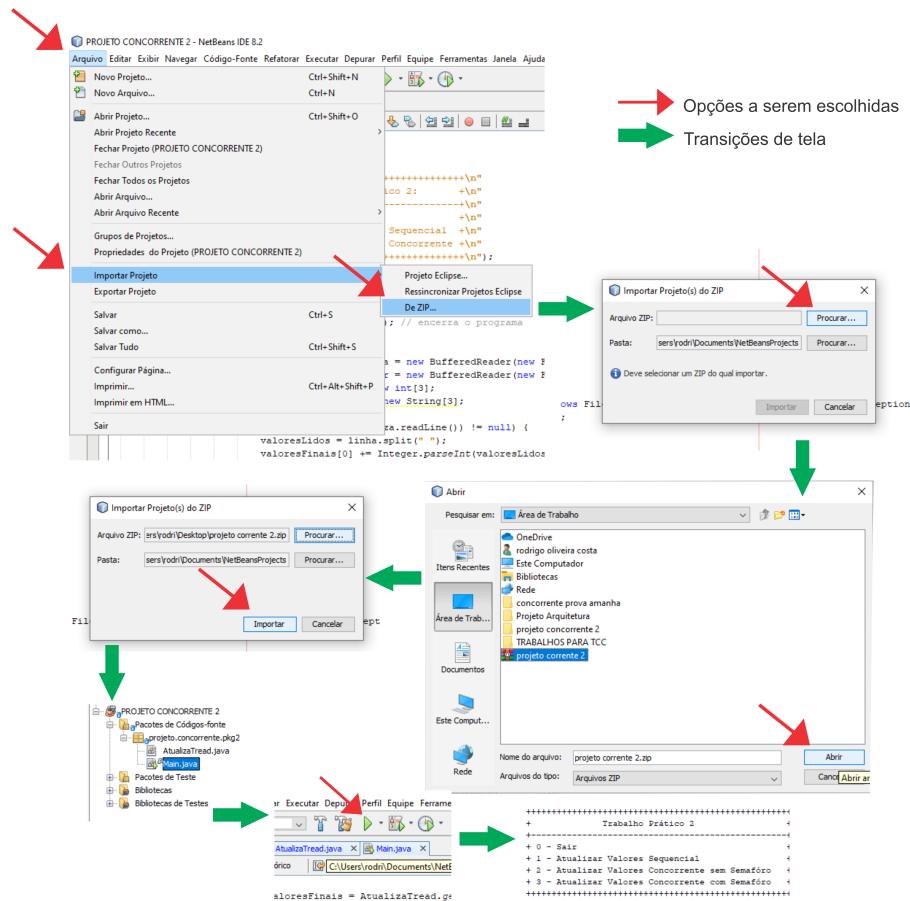


Figura 1 – Como importar um projeto no NetBeans.

Fonte: Autoria própria

Passos para executar o projeto: No *NetBeans* com o projeto aberto vá em executar projeto. E siga as opções de menu:

Atualizar Valores Sequencial

Atualizar Valores Concorrente sem Semaforo

Atualizar Valores Concorrente com Semaforo

Cada uma dessas opções terá seus respectivos resultados na tela.

3.0.2 Explicação do código-fonte

No arquivo Main.java, o método main() começa mostrando as informações necessárias (menu) para utilização do programa. Um *switch/case* decide se o programa será executado sequencialmente, concorrentemente sem semáforo ou concorrentemente com semáforo.

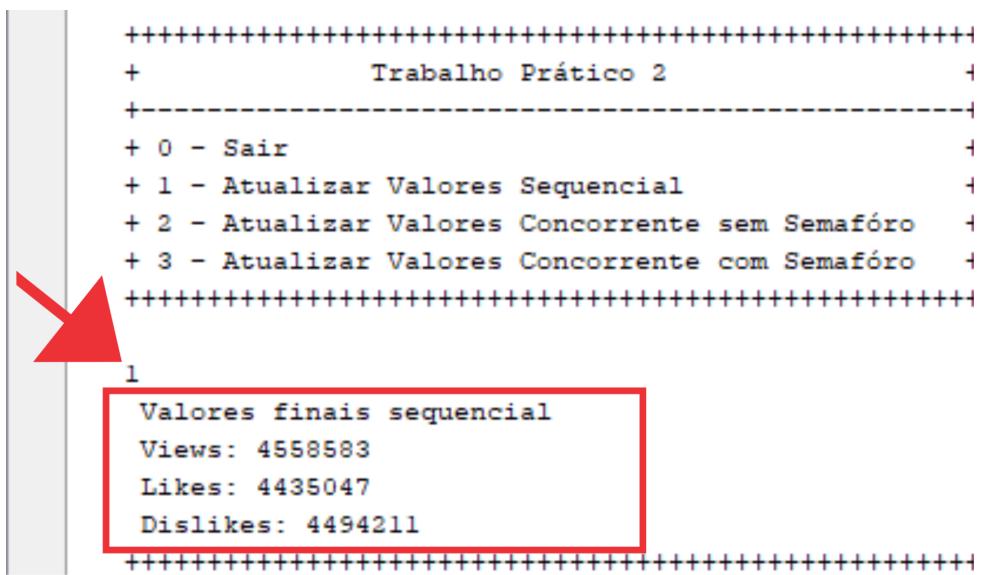
O método sequencial() lê os arquivos no formato txt. Vale ressaltar que este método possui um vetor de inteiros chamado valoresFinais. Este vetor é responsável por armazenar as informações obtidas nos arquivos referentes aos *views*, *likes* e *dislikes*, respectivamente em seus três índices. Cada linha dos arquivos são lidas sequencialmente através de dois loops, um para cada arquivo, onde o vetor valoresFinais é atualizado a cada iteração. Terminando os loops, as informações do vetor (*views*, *likes* e *dislikes*) são fornecidos na tela.

O método concorrente() é responsável por executar o programa com ou sem a utilização de semáforo. O método possui apenas um parâmetro booleano responsável por definir qual dos dois será utilizado. São utilizadas dez *threads* em ambos, porém apenas a utilização de semáforo faz com que a contagem das informações do arquivo seja feita corretamente, comparando com o programa sequencial.

As principais informações sobre a classe AtualizaTread são: ela possui um vetor de inteiros *static* parecido com o vetor utilizado na execução sequencial, um atributo do tipo *Semaphore* e um atributo inteiro chamado inicioDeLeitura responsável por controlar qual linha uma *thread* deve ler; possui também um método run() composto por um if-else, onde, caso seja utilizado semáforo, as threads são gerenciadas por ele, e caso contrário todos os métodos entram sem gerenciamento no método execucao(). O método execucao() lê 100 linhas dos arquivos atualizarValores.txt e diminuirValores.txt, e os valores do vetor *static* valoresFinais(*views*, *likes*, *dislikes*) é atualizado. Por fim, o método concorrente() printa o vetor valoresFinais, o vetor da classe AtualizaTread.

4 Resultados e Conclusão

Os resultados obtidos a partir dos algoritmos utilizados nesse trabalhos conclui que a utilização de *thread* podem causar resultados divergentes dos esperados. Sendo assim a utilização de semáforos faz as *threads* terem a propriedade de exclusão mutua. possibilitando que o algoritmo com *threads* chegue nos mesmo resultados do algoritmo sequencial. As imagem a baixo apresentam os possíveis resultados:



```
+-----+
+           Trabalho Prático 2           +
+-----+
+ 0 - Sair
+ 1 - Atualizar Valores Sequencial
+ 2 - Atualizar Valores Concorrente sem Semaforo
+ 3 - Atualizar Valores Concorrente com Semaforo
+-----+
1
Valores finais sequencial
Views: 4558583
Likes: 4435047
Dislikes: 4494211
+-----+
```

Figura 2 – Resultado do programa executado sequencialmente.

Fonte: Autoria própria

```
-----  
+++++  
+          Trabalho Prático 2          +  
+-----+  
+ 0 - Sair                         +  
+ 1 - Atualizar Valores Sequencial +  
+ 2 - Atualizar Valores Concorrente sem Semaforo +  
+ 3 - Atualizar Valores Concorrente com Semaforo +  
+++++  
2  
Valores finais com 10 Threads sem usar Semáforo  
Views: 4398507  
Likes: 4334035  
Dislikes: 4401624  
+++++
```

Figura 3 – Resultado do programa executado sem utilização de semáforo.

Fonte: Autoria própria

```
-----  
+++++  
+          Trabalho Prático 2          +  
+-----+  
+ 0 - Sair                         +  
+ 1 - Atualizar Valores Sequencial +  
+ 2 - Atualizar Valores Concorrente sem Semaforo +  
+ 3 - Atualizar Valores Concorrente com Semaforo +  
+++++  
3  
Valores finais com 10 Threads usando Semáforo  
Views: 4558583  
Likes: 4435047  
Dislikes: 4494211  
+++++
```

Figura 4 – Resultado do programa executado com utilização de semáforo.

Fonte: Autoria própria

Referências

BEN, M. **Principles of Concurrent and Distributed Programming, Second Edition.** Second. [S.l.]: Addison-Wesley, 2006. ISBN 9780321312839.

BURCKHARDT, S. et al. Cloud types for eventual consistency. In: SPRINGER. European Conference on Object-Oriented Programming. [S.l.], 2012. p. 283–307.

ORACLE. **Thread documentation in Java.** 2019. Disponível em: <<https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>>. Acesso em: 2 dez. 2019.

SCOTT, T. *Why Computers Can't Count Sometimes.* 2017. Disponível em: <https://www.youtube.com/watch?v=RY_2gElt3SA>.