Laboratório de Programação Concorrente

Lab7

Descrição Geral

Neste laboratório, iremos desenvolver um **sistema de processamento de pedidos** para uma loja de e-commerce com foco em **concorrência e controle de acesso seguro a dados compartilhados**. O sistema deve simular múltiplos clientes, que fazem pedidos simultâneos, e workers (threads), que processam esses pedidos. O desafio será garantir que os pedidos sejam processados de forma eficiente e correta, evitando problemas de concorrência, como condições de corrida e inconsistência de dados.

A solução exigirá a utilização das estruturas e técnicas avançadas de concorrência da linguagem Java, especificamente o pacote java.util.concurrent, para gerenciar filas de pedidos, controlar o acesso ao estoque e garantir a integridade dos dados. É importante destacar que este pacote disponibiliza diversas estruturas além das exemplificadas em sala de aula, por isso, identifiquem as necessidades do sistema e quais as estruturas mais adequadas para auxiliar no desenvolvimento da solução.

Este laboratório será desenvolvido em times de **até 5 pessoas.**

Objetivo

- Gerenciar múltiplos pedidos simultâneos feitos por diferentes clientes.
- Controlar o acesso ao estoque compartilhado de produtos, permitindo que múltiplos trabalhadores processem pedidos simultaneamente sem causar inconsistência nos dados.
- Simular cenários de concorrência real, onde pedidos são feitos e processados de forma paralela e assíncrona.
- Gerar relatórios periódicos sobre o estado do sistema, como o número de pedidos processados e o valor total das vendas.

Requisitos Funcionais

- **Fila de Pedidos**
    - Todos os pedidos feitos pelos clientes devem ser colocados em uma fila concorrente, onde as threads irão retirar os pedidos para processá-los.

    - A fila deve suportar um número limitado de pedidos. Se a fila estiver cheia, novos pedidos devem ser bloqueados até que haja espaço.

- **Processamento de Pedidos**
    - Cada pedido deve conter uma lista de produtos e quantidades. Quando o pedido é processado por uma thread, o sistema deve verificar a disponibilidade de todos os itens solicitados no estoque.
    - Se os itens estiverem disponíveis, eles devem ser descontados do estoque e o pedido concluído. Caso contrário, o pedido deve ser rejeitado ou ficar pendente. No último caso, o pedido é colocado em uma fila de espera para reprocessamento após o reabastecimento do estoque.

- **Leitura e Escrita no Estoque**
    - O estoque é um recurso compartilhado que deve suportar operações de leitura e escrita concorrentes. Várias threads podem consultar o estoque simultaneamente, mas apenas uma thread pode atualizar o estoque de cada vez.
    - Leitura Simultânea: Múltiplas threads podem verificar o estoque ao mesmo tempo, desde que nenhuma esteja realizando atualizações.
    - Escrita Exclusiva: Quando uma thread estiver atualizando o estoque (por exemplo, retirando itens para um pedido), nenhuma outra operação de leitura ou escrita pode ocorrer até que a operação seja concluída.

- **Simulação de Reabastecimento Automático**
    - Periodicamente, o sistema deve reabastecer automaticamente o estoque de produtos. Essa operação deve ser feita de forma concorrente, sem bloquear as operações de leitura ou o processamento de pedidos.
    - Esse reabastecimento deve acontecer em intervalos regulares (por exemplo, a cada 10 segundos) e de forma assíncrona, sem interromper o processamento de novos pedidos.

- **Relatório de Vendas**
    - A cada 30 segundos, o sistema deve gerar um relatório com as seguintes informações:

        - Número total de pedidos processados.
        - Valor total das vendas.
        - Número de pedidos rejeitados (por falta de estoque).
    - Esse relatório deve ser gerado de forma assíncrona, sem interromper o processamento de novos pedidos.

Requisitos Não Funcionais

- Eficiência: O sistema deve ser capaz de processar pedidos de forma rápida e eficiente, mesmo sob alta carga de trabalho.
- Concorrência Segura: O sistema deve garantir que não ocorra inconsistência nos dados do estoque, mesmo com várias threads acessando o recurso simultaneamente.

Desafios Extras

- **Sistema de Pedidos Prioritários**
    - Você pode adicionar a capacidade de gerenciar pedidos prioritários, garantindo que os pedidos de maior prioridade sejam processados antes dos pedidos de menor prioridade.

- **Controle de Transações de Pagamento**
    - Você pode simular um sistema de pagamento onde os pedidos só são processados após a confirmação do pagamento. Use CompletableFuture para garantir que a confirmação do pagamento seja realizada de forma assíncrona.

- **Reabastecimento Inteligente**
    - Você pode implementar um sistema de reabastecimento inteligente, onde o estoque de produtos mais populares é reabastecido com mais frequência.

- **Aplicação distribuída**
    - Você pode implementar o sistema de tal forma que os clientes sejam representados por processos diferentes do servidor do e-commerce, ao invés de representá-los por diferentes threads.

Detalhes da entrega

O código desenvolvido será entregue através de um repositório no github. Para isso, no início da aula, crie um repositório **PRIVADO**. Adicione os professores como colaboradores (https://github.com/thiagomanel/ e https://github.com/giovannifs). E, em seguida, submeta as informações para o form (link).

Além disso, é muito importante que você faça commits frequentes. Qualquer mudança relevante em uma função, deve ser comitada.

Para fins de testes, você pode desenvolver classes utilitárias que gerem pedidos para clientes de acordo com alguma política de sua preferência, como também uma classe que gere um estoque inicial para o sistema.

Considere que o sistema deve executar da seguinte forma:

```
$ ./e_commerce_livre
Estoque abastecido com X itens de Y produtos.
Pedido a do Cliente b foi processado.
Pedido c do Cliente d foi processado.
.
.
.
Relatório de Vendas:
.
.
.
Sistema reabastecido com com Z itens de W produtos.
.
.
.
```

Ou seja, ao iniciar o sistema, as threads existentes no sistema iniciam seu trabalho da forma especificada na descrição.

Prazo

03/out/24 23:59

Appendix

https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/concurrent/package-summary.html
https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/concurrent/locks/package-summary.html