Trabalho Prático: O Desafio da Concorrência na AMATUR

Disciplina: Programação Orientada a Objetos (Java)

Prof. Jean Bertrand

Tema: Domando o Caos com Threads e Sincronização

1. Contextualização

A AMATUR está enfrentando um problema crítico em seu sistema de vendas. Em rotas muito concorridas, como Boa Vista–Manaus, múltiplos agentes de venda tentam reservar a mesma poltrona simultaneamente. O sistema atual não possui controle de concorrência, resultando em *overbooking* (venda de mais passagens do que assentos).

Seu objetivo é simular esse cenário caótico e, em seguida, implementar as soluções de segurança apresentadas em aula.

---

2. Etapas do Desenvolvimento

Parte I: O Caos (Race Condition)

Implemente o cenário inicial onde o problema acontece.

- Crie a classe `Onibus` com um atributo `int assentosDisponiveis` (inicie com 5 assentos) .
- Crie o método `reservarAssento(String agente)` que verifica se há vagas (`if > 0`), dorme por 100ms (`Thread.sleep`) para simular latência, e depois decrementa a vaga .
- Crie 7 threads de agentes tentando comprar passagens ao mesmo tempo .
- **Objetivo:** Demonstrar no console que passagens foram vendidas em duplicidade (saldo negativo ou venda excedente) .

Parte II: A Solução com Blocos Sincronizados (Mutex)

Corrija o problema de *Race Condition* utilizando a palavra-chave `synchronized`.

- Ao invés de sincronizar o método inteiro (o que reduz performance), utilize um **Bloco Synchronized** (`synchronized(this)`) apenas na seção crítica do código (a verificação e o decremento do assento).
- Garanta que a operação de verificação e venda seja atômica.

Parte III: O "Desafio de Casa" (Wait e Notify)

Implemente a funcionalidade de **Cancelamento de Passagens** conforme sugerido no slide 36 .

- Implemente o padrão **Produtor-Consumidor**:
  - Se um agente tentar vender e o ônibus estiver lotado, a thread não deve apenas falhar, ela deve entrar em espera usando `wait()`.
  - Crie uma thread separada que simula um **Cancelamento** após alguns segundos. Ao cancelar, ela incrementa o número de assentos e usa `notify()` ou `notifyAll()` para avisar as threads que estavam esperando.
- **Nota:** Lembre-se que `wait()` e `notify()` devem ser chamados dentro de um bloco sincronizado.

Parte IV: Controle de Fluxo com Semáforos

Imagine que o servidor da AMATUR só aguenta 3 conexões simultâneas de agentes, independentemente de quantos assentos restam.

- Utilize a classe `Semaphore` do pacote `java.util.concurrent`.
- Inicialize o semáforo com 3 permissões (`new Semaphore(3)`).
- Faça com que cada Agente precise adquirir (`acquire()`) uma permissão antes de tentar verificar os assentos e liberar (`release()`) ao terminar.
- Isso simula o "Segurança da Balada" controlando a entrada no sistema .

---

3. Entregáveis e Critérios de Avaliação

O trabalho deve ser entregue em um repositório Git contendo:

1. **Código Fonte:** Classes `Onibus`, `AgenteVenda`, `Cancelamento` e `Main`.
2. **Relatório (README):**
   - Print do console mostrando o erro de *Overbooking* (Parte I).
   - Explicação de como o `synchronized` resolveu a inconsistência de dados.
   - Explicação de como o `wait/notify` economiza CPU evitando *busy-waiting*