**Universidade Federal de Pelotas**

Disciplina de Sistemas Operacionais

Prof. Rafael Burlamaqui Amaral

# Etapa 1 – Simulacao do Impasse do Delivery

*Deadlock com Threads em Java usando Semaforos*

**Autores:**

Pamela Braga
Nicolas Cipriano

Pelotas, 22 de fevereiro de 2026

# Sumário

# 1 Introducao

Este relatorio descreve o desenvolvimento da Etapa 1 do trabalho pratico da disciplina de Sistemas Operacionais, cujo tema e a simulacao de um cenario de *deadlock* inspirado em um aplicativo ficticio de entregas chamado *Laranjal Foods*.

O objetivo principal foi implementar, em linguagem Java, utilizando a biblioteca `java.util.concurrent.Semaphore`, um programa multithread que reproduza as condicoes classicas para a ocorrencia de um impasse (*deadlock*) e, em seguida, demonstre sua resolucao em tempo de execucao.

O trabalho esta organizado em duas versoes do programa:

- **Deadlock**: versao que forca e demonstra o travamento das threads;
- **Trylock**: versao com mecanismo de deteccao e recuperacao do *deadlock* via `tryAcquire` com *timeout*.

# 2 Descricao do Problema

O cenario simula o aplicativo *Laranjal Foods*, em que cada restaurante possui uma moto exclusiva e um pedido para ser entregue. Para realizar uma entrega, o entregador precisa obter **ambos** os recursos do mesmo restaurante: a moto e o pedido.

A frota de entregadores e mista, composta por dois perfis com comportamentos distintos:

## 2.1 Entregador Veterano

O Veterano prioriza a logistica. Seu fluxo de execucao e:

1. Escolhe aleatoriamente um restaurante $i$;
2. Tenta adquirir a **moto** do restaurante $i$;
3. Simula o tempo de caminhada ate o balcao (*sleep*);
4. Tenta adquirir o **pedido** do restaurante $i$;
5. Realiza a entrega e libera os dois recursos.

## 2.2 Entregador Novato

O Novato e ansioso para garantir a comissao. Seu fluxo de execucao e:

1. Escolhe aleatoriamente um restaurante $i$;
2. Tenta adquirir o **pedido** do restaurante $i$;
3. Simula o tempo de caminhada ate o estacionamento (*sleep*);
4. Tenta adquirir a **moto** do restaurante $i$;
5. Realiza a entrega e libera os dois recursos.

### 2.3 A Situacao de Deadlock

O *deadlock* ocorre especificamente quando um Veterano e um Novato escolhem o mesmo restaurante quase simultaneamente:

1. O Veterano adquire a chave da moto do restaurante $i$ e vai buscar o pedido;
2. O Novato adquire o pedido do restaurante $i$ e vai buscar a moto;
3. Resultado: o Veterano aguarda o pedido (que o Novato segura) e o Novato aguarda a moto (que o Veterano segura). Ambos ficam bloqueados indefinidamente.

Essa situacao satisfaz as quatro condicoes de Coffman para um *deadlock*: **exclusao mutua**, **posse e espera**, **nao-preempcao** e **espera circular**.

## 3 Implementacao

### 3.1 Estrutura do Projeto

O projeto foi implementado em Java e esta organizado da seguinte forma:

```
Etapa_1/
  Deadlock/
    Main.java
    Senior.java
    Rookie.java
  Trylock/
    Main.java
    Senior.java
    Rookie.java
```

### 3.2 Recursos Compartilhados

Os recursos de cada restaurante sao representados por arrays de semaforos binarios (equivalentes a mutexes), criados em `Main.java`:

```
1 Semaphore[] motos   = new Semaphore[N];
2 Semaphore[] pedidos = new Semaphore[N];
3
4 for (int l = 0; l < N; l++) {
5     motos[l]   = new Semaphore(1);
6     pedidos[l] = new Semaphore(1);
7 }
```

O numero de restaurantes $N$ e definido pelo usuario (entre 5 e 10). O numero de threads de cada tipo e $N + 2$, garantindo que ha mais entregadores do que restaurantes, o que aumenta a probabilidade de conflito e favorece a ocorrencia do *deadlock*.

Vale destacar que, conforme regra obrigatoria do enunciado, em todo laco de repeticao utiliza-se a variavel `l` em vez de `i`.

## 3.3 Threads: Veterano e Novato

Cada tipo de entregador e implementado como uma classe que implementa `Runnable`. A ordem de aquisicao dos semaforos e inversa entre os dois tipos, criando a condicao de espera circular.

### 3.3.1 Senior.java (Veterano)

```
1 // Pega moto primeiro, depois pedido
2 motos[l].acquire();
3 Thread.sleep(1000); // janela para o deadlock
4 pedidos[l].acquire();
```

### 3.3.2 Rookie.java (Novato)

```
1 // Pega pedido primeiro, depois moto
2 pedidos[l].acquire();
3 Thread.sleep(1000); // janela para o deadlock
4 motos[l].acquire();
```

O `Thread.sleep(1000)` entre as duas aquisicoes e fundamental: ele cria uma janela de tempo suficiente para que a outra thread adquira o recurso oposto, tornando o *deadlock* praticamente inevitavel quando dois entregadores escolhem o mesmo restaurante.

# 4 Versao com Deadlock

## 4.1 Descricao

Na versao `Deadlock/`, as threads utilizam `acquire()` padrao, que bloqueia indefinidamente. Quando o *deadlock* ocorre, as threads envolvidas ficam presas para sempre, sem nenhuma mensagem adicional sobre o restaurante em questao.

## 4.2 Saida Observada

A Figura 1 mostra a saida do terminal durante a execucao com $N = 5$ restaurantes. E possivel observar que diversas threads entram em estado de espera com a mensagem `Aguardando...` e nunca mais aparecem no log. Isso confirma o *deadlock*: os recursos estao mutuamente bloqueados e nenhuma mensagem de conclusao de entrega e gerada para os restaurantes afetados.

```
[Novato 4]: Tentando pegar o pedido do Restaurante 1...
[Veterano 1]: Tentando pegar a moto do Restaurante 1...
[Veterano 3]: Tentando pegar a moto do Restaurante 2...
[Veterano 0]: Peguei a chave da moto do Restaurante 4!
[Novato 3]: Tentando pegar o pedido do Restaurante 2...
[Veterano 1]: Peguei a chave da moto do Restaurante 1!
[Veterano 3]: Peguei a chave da moto do Restaurante 2!
[Novato 3]: Peguei o pedido do Restaurante 2!
[Novato 1]: Tentando pegar o pedido do Restaurante 0...
[Veterano 6]: Tentando pegar a moto do Restaurante 4...
[Veterano 2]: Tentando pegar a moto do Restaurante 0...
[Novato 1]: Peguei o pedido do Restaurante 0!
[Novato 6]: Aguardando a moto do Restaurante 3...
[Veterano 5]: Aguardando o pedido do Restaurante 0...
[Veterano 3]: Aguardando o pedido do Restaurante 2...
[Novato 1]: Aguardando a moto do Restaurante 0...
[Veterano 0]: Aguardando o pedido do Restaurante 4...
[Novato 6]: Peguei a chave da moto do Restaurante 3!
[Novato 3]: Aguardando a moto do Restaurante 2...
[Veterano 0]: Peguei o pedido do Restaurante 4!
[Novato 6]: Realizando entrega do Restaurante 3...
[Novato 2]: Aguardando a moto do Restaurante 1...
[Veterano 1]: Aguardando o pedido do Restaurante 1...
[Veterano 0]: Realizando entrega do Restaurante 4...
[Veterano 6]: Peguei a chave da moto do Restaurante 4!
[Novato 6]: Entrega do Restaurante 3 concluída!
[Veterano 0]: Entrega do Restaurante 4 concluída!
[Veterano 0]: Tentando pegar a moto do Restaurante 3...
[Veterano 0]: Peguei a chave da moto do Restaurante 3!
[Novato 6]: Tentando pegar o pedido do Restaurante 2...
[Veterano 6]: Aguardando o pedido do Restaurante 4...
[Veterano 6]: Peguei o pedido do Restaurante 4!
[Veterano 6]: Realizando entrega do Restaurante 4...
[Veterano 0]: Aguardando o pedido do Restaurante 3...
[Veterano 0]: Peguei o pedido do Restaurante 3!
[Veterano 0]: Realizando entrega do Restaurante 3...
[Veterano 6]: Entrega do Restaurante 4 concluída!
[Veterano 0]: Entrega do Restaurante 3 concluída!
[Veterano 6]: Tentando pegar a moto do Restaurante 0...
[Veterano 0]: Tentando pegar a moto do Restaurante 1...
```

Figura 1: Saida do terminal – versao com Deadlock

# 5 Versao com Resolucao (Trylock)

## 5.1 Descricao

Na versao `Trylock/`, as threads utilizam `tryAcquire(2, TimeUnit.SECONDS)` na tentativa de obter o segundo recurso. Caso o *timeout* expire, a thread detecta o possivel *deadlock*, libera o primeiro recurso que ja estava segurando e recomeça o ciclo apos uma pequena pausa.

### 5.1.1 Senior.java com Trylock

```
1 motos[l].acquire(); // pega a moto
2 Thread.sleep(1000);
3
4 boolean pegouPedido = pedidos[l].tryAcquire(2, TimeUnit.SECONDS);
5 if (!pegouPedido) {
6     System.out.println("[Veterano " + id + "]: DEADLOCK DETECTADO! "
```

```
        + "Soltando moto do Restaurante " + l + " e recome ando...");
    motos[l].release();
    Thread.sleep(500);
    continue;
}
```

### 5.1.2 Rookie.java com Trylock

```
pedidos[l].acquire(); // pega o pedido
Thread.sleep(1000);

boolean pegouMoto = motos[l].tryAcquire(2, TimeUnit.SECONDS);
if (!pegouMoto) {
    System.out.println("[Novato " + id + "]: DEADLOCK DETECTADO! "
        + "Soltando pedido do Restaurante " + l + " e recome ando...");
    pedidos[l].release();
    Thread.sleep(500);
    continue;
}
```

## 5.2 Saida Observada

A Figura 2 apresenta a execucao da versao com resolucao de *deadlock*. Nesta versao, e possivel observar as mensagens de **DEADLOCK DETECTADO!** sendo emitidas por multiplas threads. Apos detectar o impasse, cada thread libera o recurso que segurava e recomeça com um novo restaurante aleatorio. O sistema se recupera e as entregas continuam sendo concluidas normalmente.

```
Número de restaurantes (5 a 10): 5
[Novato 1]: Tentando pegar pedido do Restaurante 4...
[Novato 6]: Tentando pegar pedido do Restaurante 3...
[Veterano 6]: Tentando pegar a moto do Restaurante 4...
[Novato 0]: Tentando pegar pedido do Restaurante 3...
[Veterano 3]: Tentando pegar a moto do Restaurante 1...
[Veterano 2]: Tentando pegar a moto do Restaurante 3...
[Novato 5]: Tentando pegar pedido do Restaurante 4...
[Veterano 3]: Peguei a chave da moto do Restaurante 1!
[Novato 3]: Tentando pegar pedido do Restaurante 1...
[Veterano 5]: Tentando pegar a moto do Restaurante 3...
[Novato 4]: Tentando pegar pedido do Restaurante 3...
[Novato 1]: Peguei o pedido do Restaurante 4!
[Veterano 4]: Tentando pegar a moto do Restaurante 1...
[Veterano 0]: Tentando pegar a moto do Restaurante 4...
[Veterano 1]: Tentando pegar a moto do Restaurante 1...
[Novato 6]: Peguei o pedido do Restaurante 3!
[Veterano 2]: Peguei a chave da moto do Restaurante 3!
[Veterano 6]: Peguei a chave da moto do Restaurante 4!
[Novato 3]: Peguei o pedido do Restaurante 1!
[Novato 2]: Tentando pegar pedido do Restaurante 4...
[Veterano 3]: Aguardando pedido do Restaurante 1...
[Veterano 2]: Aguardando pedido do Restaurante 3...
[Novato 6]: Aguardando moto do Restaurante 3...
[Novato 1]: Aguardando moto do Restaurante 4...
[Veterano 6]: Aguardando pedido do Restaurante 4...
[Novato 3]: Aguardando moto do Restaurante 1...
[Veterano 2]: ⚠️ DEADLOCK DETECTADO! Soltando moto do Restaurante 3 e recomeçando...
[Veterano 3]: ⚠️ DEADLOCK DETECTADO! Soltando moto do Restaurante 1 e recomeçando...
[Novato 1]: ⚠️ DEADLOCK DETECTADO! Soltando pedido do Restaurante 4 e recomeçando...
[Novato 6]: ⚠️ DEADLOCK DETECTADO! Soltando pedido do Restaurante 3 e recomeçando...
[Veterano 5]: Peguei a chave da moto do Restaurante 3!
[Veterano 4]: Peguei a chave da moto do Restaurante 1!
[Veterano 6]: ⚠️ DEADLOCK DETECTADO! Soltando moto do Restaurante 4 e recomeçando...
[Veterano 0]: Peguei a chave da moto do Restaurante 4!
[Novato 3]: ⚠️ DEADLOCK DETECTADO! Soltando pedido do Restaurante 1 e recomeçando...
[Novato 5]: Peguei o pedido do Restaurante 4!
[Novato 0]: Peguei o pedido do Restaurante 3!
[Veterano 2]: Tentando pegar a moto do Restaurante 4...
[Veterano 3]: Tentando pegar a moto do Restaurante 1...
[Novato 6]: Tentando pegar pedido do Restaurante 0...
```

Figura 2: Saida do terminal – versao com Trylock (resolucao de deadlock)

# 6 Analise Comparativa

Tabela 1: Comparacao entre as versoes implementadas

| **Criterio** | **Deadlock** | **Trylock** |
|---|---|---|
| Ocorrencia de deadlock | Sim | Sim (detectado) |
| Recuperacao automatica | Nao | Sim |
| Threads bloqueiam indefinidamente | Sim | Nao |
| Metodo de aquisicao | `acquire()` | `tryAcquire(timeout)` |
| Entregas concluidas | Parcialmente | Continuamente |
| Visibilidade no log | Implicita | Explicita (mensagem) |

A estrategia `tryAcquire` com *timeout* implementa uma forma de **deteccao e recuperacao** de *deadlock*. Ela nao previne o impasse em si, mas permite que o sistema

identifique a situacao e quebre o ciclo de espera ao liberar um dos recursos, possibilitando que outras threads avancem.

# 7 Consideracoes Finais

Este trabalho permitiu observar na pratica os conceitos teoricos de concorrencia e *deadlock* estudados em aula. A implementacao em Java com semaforos explicitos (`java.util.concurrent.Semaphore`) demonstrou de forma clara como a ordem de aquisicao de recursos entre threads com comportamentos distintos pode levar a um impasse classico.

A versao *Deadlock* evidenciou visualmente o problema: threads que entram em estado de espera e nunca mais progridem, tornando parte do sistema inoperante. A versao *Trylock* demonstrou uma abordagem pratica de recuperacao, onde o proprio sistema identifica o travamento e toma uma acao corretiva sem intervencao externa.

Como trabalho futuro, outras estrategias de prevencao poderiam ser exploradas, como a imposicao de uma **ordem global de aquisicao de recursos** – fazendo com que todos os entregadores, independentemente do tipo, adquiram sempre a moto antes do pedido – o que eliminaria a condicao de espera circular e preveniria o *deadlock* por completo.