

## Relatório – Problema do Jantar dos Filósofos

O problema do jantar dos filósofos envolve cinco filósofos sentados em uma mesa circular, alternando entre pensar, ter fome e comer. Para comer, cada filósofo precisa pegar dois garfos: o da sua esquerda e o da sua direita. Como os garfos estão entre dois filósofos, eles são recursos compartilhados e exclusivos.

No protocolo ingênuo — pegar primeiro o garfo da esquerda e depois o da direita — pode ocorrer um impasse (deadlock). Se todos os filósofos ficarem com fome ao mesmo tempo e pegarem simultaneamente o garfo à esquerda, nenhum terá acesso ao garfo da direita. Assim, todos ficam esperando indefinidamente, sem liberar os recursos.

Esse impasse ocorre porque todas as quatro condições de Coffman para deadlock são satisfeitas:

1. **Exclusão mútua:** cada garfo só pode ser usado por um filósofo por vez.
2. **Manter-e-esperar:** cada filósofo segura um garfo enquanto espera o outro.
3. **Não preempção:** não é possível tomar um garfo de outro filósofo à força.
4. **Espera circular:** cada filósofo espera pelo garfo que o vizinho está segurando, formando um ciclo.

## Solução proposta – Hierarquia de recursos

Para evitar o deadlock, adotamos uma hierarquia global nos recursos (garfos). Cada garfo recebe um índice. Cada filósofo deve sempre pegar primeiro o garfo de menor índice e depois o de maior índice. Com essa regra, elimina-se a condição de espera circular, pois não é possível formar um ciclo se todos solicitam os recursos em uma ordem fixa.

Ao remover a espera circular, um dos requisitos para deadlock deixa de existir e, portanto, o sistema não pode entrar em impasse.

## Pseudocódigo:

Dados:

N = 5 filósofos

Garfos 0..N-1

O garfo i fica entre os filósofos i e  $(i+1) \bmod N$

Para cada filósofo p:

left = min(garfo\_esquerda(p), garfo\_direita(p))

right = max(garfo\_esquerda(p), garfo\_direita(p))

Loop infinito:

pensar()

estado[p] <- "com fome"

adquirir(left) // espera até o garfo estar livre

adquirir(right) // espera até o garfo estar livre

estado[p] <- "comendo"

comer()

liberar(right)

liberar(left)

estado[p] <- "pensando"

# Relatório – Threads, Semáforos e Condição de Corrida

## 1. Objetivo

O experimento tem como objetivo demonstrar uma condição de corrida ao atualizar um contador compartilhado entre múltiplas threads sem exclusão mútua, e depois corrigir o problema usando um semáforo binário.

## 2. Condição de corrida (versão sem semáforo)

Na primeira implementação, diversas threads executam `count++` simultaneamente. O operador de incremento não é atômico: ele envolve leitura, soma e escrita. Quando duas threads executam essas etapas ao mesmo tempo, uma delas sobrescreve o resultado da outra.

Por isso, mesmo que o valor esperado seja  $T \times M$ , o valor obtido é menor, evidenciando a condição de corrida.

## 3. Solução com semáforo

A segunda implementação usa:

```
Semaphore sem = new Semaphore(1, true);
```

- O semáforo tem **1 permissão**, funcionando como exclusão mútua.
- A opção `true` habilita modo **fair (FIFO)**, garantindo ordem justa entre as threads.
- Cada incremento só ocorre dentro de uma seção crítica protegida por `acquire()` e `release()`.

Como apenas uma thread acessa o contador por vez, o valor final é sempre **correto**.

## 4. Consistência de memória

O `release()` estabelece uma garantia **happens-before** em relação a um futuro `acquire()` de outra thread. Isso garante visibilidade entre threads: atualizações no contador não ficam “invisíveis”.

## 5. Comparação dos resultados

- **Sem semáforo:** rápido, porém incorreto → condições de corrida.
- **Com semáforo:** correto, porém mais lento → cada thread precisa esperar a outra.

Isso mostra o trade-off clássico:

- **Desempenho vs correção e sincronização.**

A redução de throughput ocorre porque o semáforo serializa os acessos.

## 6. Conclusão

O experimento demonstra que operações aparentemente simples, como um incremento, não são seguras em ambientes concorrentes sem sincronização. O uso de `Semaphore(1, true)` elimina a condição de corrida, garante justiça e assegura o valor correto do contador, confirmando a necessidade de exclusão mútua em atualizações compartilhadas.

## Relatório – Deadlock com Dois Locks

### 1. Descrição do cenário

O experimento utiliza duas threads e dois locks (LOCK\_A e LOCK\_B).

A Thread 1 adquire LOCK\_A e depois tenta adquirir LOCK\_B.

A Thread 2 adquire LOCK\_B e depois tenta adquirir LOCK\_A.

Essa ordem inversa de aquisição gera uma espera circular clássica:

cada thread segura um lock e espera pelo lock que a outra possui.

O programa não lança exceção, mas simplesmente deixa de progredir — caracterizando deadlock.

### 2. Condições de Coffman presentes

O deadlock acontece porque **todas** as quatro condições necessárias estão presentes:

1. **Exclusão mútua:** Cada lock só pode ser adquirido por uma thread por vez.
2. **Manter-e-esperar:** Cada thread segura um lock enquanto aguarda o outro.
3. **Não preempção:** Locks não podem ser tomados de outra thread à força; só são liberados voluntariamente.
4. **Espera circular:** Thread 1 espera por B → Thread 2 espera por A.  
Isso forma um ciclo fechado.

Como todas as quatro condições se manifestam, o deadlock ocorre.

### 3. Correção – Hierarquia de Recursos

Para evitar o deadlock, aplicamos uma ordem global de aquisição de locks. A solução é exigir que todas as threads adquiram os locks na mesma ordem:

Sempre adquirir primeiro LOCK\_A, depois LOCK\_B.

Isso remove a condição de espera circular, pois não pode existir mais um ciclo se todos seguem a mesma hierarquia.

## Código corrigido

```
public class DeadlockFixed {  
    static final Object LOCK_A = new Object();  
    static final Object LOCK_B = new Object();  
  
    public static void main(String[] args) {  
        Runnable r = () -> {  
            synchronized (LOCK_A) {  
                dormir(50);  
                synchronized (LOCK_B) {  
                    System.out.println(Thread.currentThread().getName() + " concluiu");  
                }  
            }  
        };  
  
        Thread t1 = new Thread(r, "T1");  
        Thread t2 = new Thread(r, "T2");  
  
        t1.start();  
        t2.start();  
    }  
  
    static void dormir(long ms) {  
        try { Thread.sleep(ms); }  
        catch (InterruptedException e) { Thread.currentThread().interrupt(); }  
    }  
}
```

## 5. Conclusão

O deadlock ocorre porque todas as condições de Coffman são satisfeitas. A correção baseada em hierarquia de recursos elimina a espera circular, impedindo que threads adquiram locks em ordem invertida. Essa estratégia corresponde ao

mesmo princípio usado na solução do Jantar dos Filósofos, onde a ordem fixa dos garfos impede a formação de ciclos de espera.