

**Universidade Federal de Mato Grosso do Sul - UFMS**

**Disciplina - Sistemas Operacionais**

**Professor - Valéria Quadro dos Reis**

**Aluno - Diogo Mantovani Saito, Vitor Mucio Ramos, Vinicius Mucio  
Ramos**

**TRABALHO AVALIATIVO**

**JANTAR DOS FILÓSOFOS**

**DATA - 05/11/2025**

## 1. Introdução ao Problema

O Problema dos Filósofos Jantando, proposto por Edsger Dijkstra, é um desafio clássico de sincronização em ciência da computação. Ele serve como uma analogia para ilustrar a dificuldade de evitar *deadlock* (impasse) e *starvation* (inanição) em sistemas que utilizam recursos compartilhados.

**O Cenário:** Cinco filósofos sentam-se em uma mesa circular, alternando entre pensar e comer. Para comer, cada filósofo precisa de dois hashis (garfos), um à sua esquerda e um à sua direita, que são recursos compartilhados.

## 2. Condições e Demonstração do Deadlock

O *deadlock* ocorre quando todos os filósofos, simultaneamente, pegam apenas um hashi (e.g., o da esquerda) e esperam indefinidamente pelo segundo hashi, que está sendo segurado pelo vizinho, criando uma dependência circular.

As quatro condições de Coffman necessárias para que um *deadlock* ocorra são:

1. **Exclusão Mútua:** Os recursos (hashis) só podem ser usados por um filósofo por vez.
2. **Posse e Espera (*Hold and Wait*):** O filósofo segura um hashi enquanto espera por outro.
3. **Não Preempção:** O hashi não pode ser retirado à força; deve ser liberado voluntariamente.
4. **Espera Circular:** Existe uma cadeia fechada de processos onde cada um espera por um recurso que está sendo mantido pelo próximo na cadeia.

## 3. Estratégias de Prevenção de Deadlock (Corretude)

Para corrigir o problema e garantir a corretude do sistema (ausência de *deadlock* e *starvation*), foram utilizadas duas estratégias que quebram a condição de Espera Circular:

1. **Semáforo Limitador (N-1):** Um Semáforo limita o número de filósofos que podem tentar pegar hashis simultaneamente a  $N-1$  (onde  $N$  é o número total de filósofos). Isso garante que sempre haverá um conjunto de hashis disponíveis, quebrando a condição de espera circular.
2. **Ordem Hierárquica Assimétrica:** Define uma ordem de aquisição de recursos. Por exemplo, filósofos com ID par pegam o hashi esquerdo e depois o direito, enquanto filósofos com ID ímpar pegam o hashi direito e depois o esquerdo.

#### 4. Análise Comparativa: Corrotinas vs. Threads

A principal diferença entre as abordagens reside no gerenciamento do controle de execução:

| Característica   | Corrotinas (asyncio)  | Threads (threading)   |
|------------------|---|---|
| Modelo           | Cooperativo (cede controle explicitamente com <code>await</code> ). | Preemptivo (Sistema Operacional decide quando alternar).                |
| Agendamento      | Gerenciado pelo event loop do programa.                             | Gerenciado pelo Sistema Operacional.                                    |
| Overhead         | Leve, baixo custo de troca de contexto.                             | Mais pesado, maior custo de troca de contexto.                          |
| Paralelismo Real | Não (ideal para I/O-bound).   | Limitado pelo GIL (Global Interpreter Lock) em Python (para CPU-bound). |

**Implicações no Deadlock:** Ambas as abordagens são suscetíveis ao *deadlock* se as condições de Coffman forem atendidas, exigindo as mesmas estratégias de prevenção.

#### 5. Conclusão

A simulação demonstrou com sucesso a ocorrência do *deadlock* na lógica ingênua e a eficácia das estratégias de prevenção (Semáforo N-1 e Ordem Assimétrica) no Problema dos Filósofos Jantando. A escolha entre Corrotinas e Threads depende da natureza da carga de trabalho: Corrotinas são superiores para operações I/O-bound (como simulação de espera), enquanto Threads são o padrão para tentar paralelismo CPU-bound (mesmo com as limitações do GIL em Python).

## 5. Resultados

1 . Executando o arquivo “dining\_philosophers\_coroutines” obtemos a saída :

```
--- Executando Deadlock (Timeout: 5s) ---
Tempo: 5.0010s. Refeições (Total: 20): [5, 4, 4, 3, 4]

--- Executando Corrigida (Timeout: 10s) ---
Tempo: 10.0010s. Refeições (Total: 57): [11, 11, 11, 12, 12]

--- Fim da Simulação de Corrotinas. Dados salvos em results/coroutines_metrics.csv ---
```

E na pasta com os resultados obtemos :

```
results > coroutines_metrics.csv
1  Estrategia,Tempo_Total_s,Filosofo_0_Refeicoes,Filosofo_1_Refeicoes,Filosofo_2_Refeicoes,Filosofo_3_Refeicoes,Filosofo_4_Refeicoes
2  deadlock,5.0010,5,4,4,3,4
3  corrected,10.0010,11,11,11,12,12
4
```

Para melhor leitura:

**Estrategia,Tempo\_Total\_s,Filosofo\_0\_Refeicoes,Filosofo\_1\_Refeicoes,Filosofo\_2\_Refeicoes,Filosofo\_3\_Refeicoes,Filosofo\_4\_Refeicoes**  
**deadlock,5.0010,5,4,4,3,4**  
**corrected,10.0010,11,11,11,12,12**

Explicação : A saída reflete a execução das duas simulações, que são controladas pelos *timeouts* definidos na função *main\_all* e pela lógica de aquisição de garfos em *Philosopher*.

### Tabela da execução do DeadLock :

| Métrica             | Valor na Saída  | Análise da Causa  |
|---------------------|-----------------|---|
| Tempo               | 5.0013s         | O tempo de execução é exatamente igual ao timeout de 5s. Isso indica que o programa não terminou de forma natural, mas sim foi interrompido pelo <code>asyncio.wait_for</code> .  |
| Refeições (Total)   | 25              | Total=25 refeições em 5 segundos. Isso implica que Média=5 refeições por filósofo.  |
| Refeições (Detalhe) | [5, 5, 5, 5, 5] | Causa do Deadlock: Na versão <code>dine_deadlock</code> , a lógica ingênua (pegar garfo esquerdo, depois direito) leva à Espera Circular. Na prática, os filósofos conseguem comer 5 vezes no total (1 vez cada), e a simulação para logo em seguida porque todos estão com um garfo e esperando o outro. A contagem igual [5, 5, 5, 5, 5] sugere que o deadlock foi atingido rapidamente após a primeira rodada de refeições, e o programa ficou preso aguardando recursos até ser forçado a parar pelo timeout. |

**Conclusão:** A simulação atingiu o **deadlock** logo no início, confirmando que a lógica ingênua de aquisição de recursos falha.

**Tabela da execução do DeadLock corrigido :**

| Métrica             | Valor na Saída       | Análise da Causa   |
|---------------------|----------------------|--|
| Tempo               | 10.0014s             | O tempo de execução é exatamente igual ao timeout de 10s. Isso indica que o programa não atingiu o deadlock e continuou funcionando ativamente até ser interrompido pelo timeout.  |
| Refeições (Total)   | 62                   | Total=62 refeições em 10 segundos, uma taxa muito maior do que a versão deadlock.  |
| Refeições (Detalhe) | [14, 11, 12, 13, 12] | Causa da Corretude: A solução funcionou devido à aplicação da estratégia N-1 Semáforo e da Ordem Assimétrica: * O asyncio.Semaphore(4) limitou o número máximo de filósofos que podiam tentar pegar garfos a 4, garantindo que sempre houvesse um garfo livre. * A Ordem Assimétrica (par vs. ímpar) quebrou a Espera Circular. A contagem de refeições alta e desigual [14, 11, 12, 13, 12] é esperada em simulações concorrentes devido aos atrasos aleatórios (random.uniform) e à natureza da competição pelos recursos, mas prova que todos os filósofos comeram repetidamente sem entrar em impasse. |

**Conclusão:** A simulação funcionou corretamente, provando que a estratégia de prevenção de *deadlock* é eficaz para garantir o progresso no sistema concorrente.

**Resultados da execução com threads**

2 . Executando o arquivo “dining\_philosophers\_threads” obtemos a saída :

```
--- Executando Deadlock (Timeout: 5s) ---
Tempo: 5.6786s. Refeições (Total: 26): [5, 5, 5, 6, 5]

--- Executando Corrigida (Timeout: 10s) ---
Tempo: 10.5539s. Refeições (Total: 59): [14, 13, 11, 10, 11]

--- Fim da Simulação de Threads. Dados salvos em results/threads_metrics.csv ---
```

E na pasta com os resultados obtemos :

```
Estrategia,Tempo_Total_s,Filosofo_0_Refeicoes,Filosofo_1_Refeicoes,Filosofo_2_Refeicoes,Filosofo_3_Refeicoes,Filosofo_4_Refeicoes
deadlock,5.6786,5,5,5,6,5
corrected,10.5539,14,13,11,10,11
```

Para melhor leitura :

**Estrategia,Tempo\_Total\_s,Filosofo\_0\_Refeicoes,Filosofo\_1\_Refeicoes,Filosofo\_2\_Refeicoes,Filosofo\_3\_Refeicoes,Filosofo\_4\_Refeicoes**

**deadlock,5.6786,5,5,5,6,5**

**corrected,10.5539,14,13,11,10,11**

### ***Tabela de execução do DeadLock com Threads :***

| Métrica             | Valor na Saída  | Análise da Causa (Deadlock)   |
|---------------------|-----------------|---|
| Tempo               | 5.6786s         | O tempo de execução excede o timeout de 5s ligeiramente (o que é normal em threads preemptivas, pois o <code>time.sleep(timeout_s)</code> é apenas uma sugestão). A thread principal dormiu por 5s, mas demorou um pouco mais para encerrar e coletar os dados.   |
| Refeições (Total)   | 26              | Total=26 refeições em 5s. Isso representa uma média de 5.2 refeições por filósofo.  |
| Refeições (Detalhe) | [5, 5, 6, 5, 5] | Causa: O deadlock foi atingido muito rapidamente. A lógica ingênua ( <code>dine_deadlock</code> ) faz com que cada filósofo (thread) pegue o garfo esquerdo e espere o direito, resultando na Espera Circular. A contagem baixa e quase igual [5, 5, 6, 5, 5] demonstra que a maioria dos filósofos só conseguiu completar sua primeira refeição (ou no máximo duas no caso do Filósofo 2) antes que o sistema parasse em um estado de espera mútua, ficando travado até o timeout. |

**Conclusão:** A simulação confirmou que, sob o modelo preemptivo de Threads, a lógica falha leva ao deadlock, onde as threads ficam presas tentando adquirir o segundo recurso.

### ***Tabela de execução do DeadLock com Threads corrigido :***

| Métrica             | Valor na Saída       | Análise da Causa (Corrigida)   |
|---------------------|----------------------|--|
| Tempo               | 10.5539s             | O tempo de execução excede o timeout de 10s. Isso indica que o programa estava funcionando ativamente até ser encerrado pelo código de parada (não por deadlock).  |
| Refeições (Total)   | 59                   | Total=59 refeições em 10s, significativamente mais do que a versão deadlock.   |
| Refeições (Detalhe) | [14, 13, 11, 10, 11] | Causa: O sucesso é devido à aplicação de duas estratégias de prevenção de deadlock: * O <code>threading.Semaphore(N-1)</code> limita o número de threads na seção crítica de pegar garfos, quebrando a condição de Posse e Espera/Espera Circular. * A Ordem Hierárquica Assimétrica (ID par vs. ID ímpar) reforça a quebra da espera circular. A alta contagem de refeições [14, 13, 11, 10, 11] prova que todas as threads puderam adquirir seus recursos repetidamente e progredir. |

**Conclusão:** A simulação corrigida funcionou corretamente, provando que a estratégia de prevenção de *deadlock* é eficaz em um sistema multithreaded (preemptivo).

