Apresentação de performance TD3

1. Parte 1: O Jantar dos Filósofos

1.1 Dinâmica e Problema do Impasse

O problema do Jantar dos Filósofos modela a gestão de recursos limitados em sistemas concorrentes. Cinco filósofos alternam entre pensar e comer, necessitando de dois garfos (recursos compartilhados) adjacentes para se alimentar1.

No protocolo ingênuo, onde cada filósofo pega primeiro o garfo à sua esquerda e depois o da direita, surge uma situação de **impasse (deadlock)** se todos decidirem comer simultaneamente. Nesse cenário, cada filósofo segura o garfo esquerdo e aguarda indefinidamente pelo garfo direito, que está em posse do vizinho. Isso satisfaz as quatro condições de Coffman para deadlock, especificamente a **espera circular**2.

1.2 Solução Proposta: Hierarquia de Recursos

Para resolver o impasse, a solução escolhida foi a **Hierarquia de Recursos**. Esta estratégia nega a condição de **espera circular** de Coffman3.

A lógica consiste em numerar os garfos de 0 a $N-1$ e impor uma regra global: todo filósofo deve requisitar primeiro o garfo de **menor índice** e, em seguida, o de **maior índice**4. Isso impede o fechamento do ciclo, pois o último filósofo (que estaria entre o garfo $N-1$ e 0) será forçado a tentar pegar o garfo 0 primeiro (que já estaria ocupado pelo primeiro filósofo), impedindo-o de pegar o garfo $N-1$ e travar a mesa.

1.3 Pseudocódigo do Protocolo
Abaixo, o algoritmo que implementa a ordenação parcial de recursos para garantir progresso e justiça:

```
Constantes: N = 5
Recursos: Garfos[0..4]

Para cada Filósofo(i):
  // Define a ordem de pegar os garfos (menor primeiro)
  primeiro_garfo = min(i, (i + 1) % N)
  segundo_garfo = max(i, (i + 1) % N)

  Loop Infinito:
    pensar()
    estado[i] = "COM FOME"

    adquirir(Garfos[primeiro_garfo])  // Pega o menor índice
    adquirir(Garfos[segundo_garfo])   // Pega o maior índice

    estado[i] = "COMENDO"
    comer()

    liberar(Garfos[segundo_garfo])
    liberar(Garfos[primeiro_garfo])
```

```
    estado[i] = "PENSANDO"
```

2. Parte 2: Threads e Semáforos (Contador Concorrente)
2.1 Condição de Corrida

Na execução da classe `CorridaSemControle`, observou-se que o valor final do contador foi inferior ao esperado ($T \times M$). Isso ocorre devido a uma **condição de corrida**: a operação de incremento (`count++`) não é atômica (envolve leitura, modificação e escrita). Sem sincronização, múltiplas threads leem o mesmo valor antigo, incrementam e sobrescrevem o resultado uma da outra, perdendo atualizações5555.

2.2 Solução com Semáforo
A correção foi implementada na classe `CorridaComSemaphore` utilizando um `Semaphore(1, true)`.
•
Exclusão Mútua: O semáforo binário (inicializado com 1 permissão) garante que apenas uma thread entre na seção crítica (incremento) por vez6666.

•
Justiça (Fairness): O parâmetro `true` no construtor ativa o modo justo, organizando as threads em uma fila FIFO (First-In, First-Out), evitando inanição (starvation) de threads específicas7777.

2.3 Discussão de Trade-offs
O uso do semáforo garantiu a corretude do resultado (atingindo o valor exato de $T \times M$), garantindo a relação *happens-before* entre a liberação de uma thread e a aquisição de outra8. No entanto, houve uma redução significativa no *throughput* (aumento do tempo de execução) em comparação à versão sem controle, devido ao *overhead* de gerenciamento do bloqueio e trocas de contexto forçadas pela serialização do acesso.

3. Parte 3: Deadlock
3.1 Reprodução e Análise (Coffman)

O código `DeadlockDemo` reproduziu um travamento total ao utilizar duas threads e dois objetos de bloqueio (`LOCK_A` e `LOCK_B`) adquiridos em ordens inversas. As quatro condições de Coffman foram observadas9999:

1. **Exclusão Mútua:** Os locks são exclusivos.
2. **Manter-e-Esperar (Hold and Wait):** A Thread 1 segura A e espera B; a Thread 2 segura B e espera A.
3. **Não Preempção:** Uma thread não pode tomar o lock da outra à força.
4. **Espera Circular:** Criou-se um ciclo de dependência $T1 \to B \to T2 \to A \to T1$.

3.2 Correção Implementada

A correção seguiu a estratégia de **Ordenação Global de Recursos** (similar à Parte 1). O código foi alterado para garantir que **todas** as threads adquiram os locks na mesma ordem (sempre `LOCK_A` e depois `LOCK_B`)10.

Ao impor essa hierarquia, eliminamos a condição de **Espera Circular**. Se a Thread 1 pegar o `LOCK_A`, a Thread 2 será bloqueada já na tentativa de pegar o `LOCK_A` (seu primeiro passo), impedindo que ela segure o `LOCK_B` enquanto espera. Assim, o ciclo de espera torna-se impossível.

Referências:

1. https://en.wikipedia.org/wiki/Dining_philosophers_problem
2. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html
3. https://en.wikipedia.org/wiki/Deadlock_(computer_science)
4. https://www.youtube.com/watch?v=NbwbQQB7xNQ
5. https://www.geeksforgeeks.org/operating-systems/introduction-of-deadlock-in-operating-system/
6. https://www.scaler.com/topics/operating-system/dining-philosophers-problem-in-os/
7. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html
8. https://www.linkedin.com/pulse/dining-philosopher-problem-explanation-implementation-michael-putong-kjmnc
9. https://techvidvan.com/tutorials/semaphore-in-java/
10. https://www.theknowledgeacademy.com/blog/deadlock-in-os/
11. https://www.mathworks.com/help/simevents/ug/dining-philosophers-problem.html
12. https://davidvlijmincx.com/posts/how-to-use-java-semaphore/
13. https://dev.to/aryan_shourie/deadlocks-in-operating-systems-5g4o
14. https://lass.cs.umass.edu/~shenoy/courses/fall13/lectures/Lec10_notes.pdf
15. https://www.reddit.com/r/learnprogramming/comments/1ce8wyc/dining_philosophers_deadlock_problem/
16. https://www.geeksforgeeks.org/operating-systems/dining-philosophers-problem/
17. https://www.youtube.com/watch?v=FYUi-u7UWgw
18. https://www.geeksforgeeks.org/operating-systems/dining-philosopher-problem-using-semaphores/
19. https://www.geeksforgeeks.org/java/semaphore-in-java/
20. https://www.tutorialspoint.com/operating_system/introduction_to_deadlock_in_operating_system.htm

Meu roteiro:

Roteiro