Parte 1 — Filósofos

O Jantar dos Filósofos modela cinco filósofos sentados em uma mesa circular, alternando entre pensar e comer, onde para comer cada filósofo precisa dos dois garfos à sua esquerda e direita, compartilhados com os vizinhos, expondo problemas clássicos de exclusão mútua, impasse e inanição em sistemas concorrentes. Um impasse ocorre se todos pegarem simultaneamente o garfo da esquerda e aguardarem o da direita, pois ninguém progride e todos esperam indefinidamente, caracterizando deadlock no protocolo ingênuo de "pegar primeiro um garfo, depois o outro". Soluções gerais exigem negar pelo menos uma das quatro condições necessárias para deadlock — exclusão mútua, manter-e-esperar, não preempção e espera circular — sendo comum eliminar a espera circular por meio de hierarquia de recursos, usar um árbitro (garçom) ou limitar a N−1 filósofos ativos.[1]

Objetivo

- Simular o problema com N=5, registrando estados "pensando", "com fome" e "comendo", evidenciando por que o protocolo ingênuo entra em impasse.[1]
- Projetar um protocolo que evite impasse, explicando logicamente a estratégia escolhida (ex.: hierarquia de recursos ou árbitro) e por que ela quebra uma das condições para deadlock.[3][1]

Entregáveis

- Relatório curto explicando a dinâmica do problema, por que o impasse surge e qual condição para deadlock será negada na solução proposta, relacionando com as condições de Coffman.[3][1]
- Pseudocódigo ou fluxograma do protocolo, sem código em Java, demonstrando a ordem de aquisição e liberação de recursos e como a justiça/progresso é preservada.[1]

Pseudocódigo (hierarquia de recursos)

Atribua um índice único a cada garfo e force todos a requisitarem primeiro o garfo de menor índice, depois o de maior índice, removendo a espera circular ao impor uma ordem parcial fixa sobre os recursos.[3][1]

```
Dados:
  N = 5 filósofos
  Garfos 0..N-1 (garfo i fica entre filósofos i e (i+1) mod N)

Para cada filósofo p:
  left  = min(garfo_esquerda(p), garfo_direita(p))
```

```
  right = max(garfo_esquerda(p), garfo_direita(p))

Loop:
  pensar()
  estado[p] <- "com fome"
  adquirir(left)     // bloqueia até o garfo estar livre
  adquirir(right)    // bloqueia até o garfo estar livre
  estado[p] <- "comendo"
  comer()
  liberar(right)
  liberar(left)
  estado[p] <- "pensando"
```

O protocolo evita impasse porque elimina a condição de espera circular ao impor uma ordem global, garantindo que não possa existir um ciclo de espera entre filósofos.[3][1]

Parte 2 — Threads e semáforos

Um semáforo controla o acesso a um recurso compartilhado por meio de contagem de "permissões"; adquirir bloqueia quando não há permissões e liberar devolve permissões, podendo funcionar como exclusão mútua quando inicializado com um único alvará (binário). Em Java, a classe java.util.concurrent.Semaphore oferece modos justo e não justo, métodos acquire/release e garantias de consistência de memória do tipo happens-before entre release de um thread e acquire de outro, permitindo eliminar condições de corrida ao serializar acessos críticos.[2]

Desafio: contador concorrente

- Objetivo: demonstrar uma condição de corrida incrementando um contador compartilhado com múltiplas threads sem sincronização e, em seguida, corrigi-la com um semáforo binário, comparando resultados e tempo de execução.[2]
- Passos: crie um contador estático, lance T threads que cada uma executa M incrementos, meça o valor final com e sem semáforo, e discuta diferenças, fairness e impacto em throughput.[2]

Implementação em Java (modelo de solução)

O primeiro trecho demonstra a condição de corrida ao atualizar um inteiro compartilhado sem exclusão, e o segundo usa Semaphore(1, true) para assegurar exclusão mútua e justiça na ordem de aquisição.

```
// Versão com condição de corrida (sem sincronização)
import java.util.concurrent.*;
public class CorridaSemControle {
  static int count = 0;
  public static void main(String[] args) throws Exception {
    int T = 8, M = 250_000;
    ExecutorService pool = Executors.newFixedThreadPool(T);
    Runnable r = () -> {
      for (int i = 0; i < M; i++) {
        count++; // atualização não atômica: sujeito a perda de incrementos
      }
    };
    long t0 = System.nanoTime();
    for (int i = 0; i < T; i++) pool.submit(r);
    pool.shutdown();
    pool.awaitTermination(1, TimeUnit.MINUTES);
    long t1 = System.nanoTime();
    System.out.printf("Esperado=%d, Obtido=%d, Tempo=%.3fs%n",
        T * M, count, (t1 - t0) / 1e9);
  }
}


// Versão correta com Semaphore binário justo
import java.util.concurrent.*;
public class CorridaComSemaphore {
  static int count = 0;
  static final Semaphore sem = new Semaphore(1, true); // FIFO
  public static void main(String[] args) throws Exception {
    int T = 8, M = 250_000;
    ExecutorService pool = Executors.newFixedThreadPool(T);
    Runnable r = () -> {
      for (int i = 0; i < M; i++) {
        try {
          sem.acquire();
          count++;
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
        } finally {
          sem.release();
```

```
        }
      }
    };
    long t0 = System.nanoTime();
    for (int i = 0; i < T; i++) pool.submit(r);
    pool.shutdown();
    pool.awaitTermination(1, TimeUnit.MINUTES);
    long t1 = System.nanoTime();
    System.out.printf("Esperado=%d, Obtido=%d, Tempo=%.3fs%n",
        T * M, count, (t1 - t0) / 1e9);
  }
}
```

- O valor correto do contador é T×M na versão com semáforo porque acquire bloqueia quando a permissão é zero e release libera a seção crítica, garantindo exclusão mútua e ordenação justa quando configurado como fair.[2]
- Discuta o trade-off: semáforo elimina a condição de corrida ao custo de possível redução de throughput, mas com garantias happens-before que preservam visibilidade e ordem entre threads.[2]

Parte 3 — Deadlock

Deadlock é a situação em que um conjunto de processos ou threads não progride porque cada um aguarda uma ação do outro, tipicamente a liberação de um lock, sendo necessário que quatro condições ocorram simultaneamente: exclusão mútua, manter-e-esperar, não preempção e espera circular (Condições de Coffman). Estratégias de tratamento incluem ignorar, detectar e recuperar, prevenir quebrando uma das condições (ex.: impedir manter-e-esperar ou impor uma hierarquia), ou evitar via análise antecipada de alocações seguras (ex.: Banker's).[3]

Atividade com deadlock

- Cenário: duas threads e dois locks (A e B) tal que a Thread 1 adquire A depois B, enquanto a Thread 2 adquire B depois A, produzindo uma espera circular quando cada uma segura um lock e aguarda o outro indefinidamente.[3]
- Objetivo: reproduzir o deadlock, explicar quais condições de Coffman se manifestaram e implementar uma correção, preferencialmente impondo uma ordem global de aquisição de recursos para eliminar a espera circular (hierarquia de recursos).[1][3]

Implementação mínima (deadlock)

Este exemplo deve “travar” sem lançar exceção, evidenciando o sistema parado pela espera circular.

```
import java.util.concurrent.*;

public class DeadlockDemo {
  static final Object LOCK_A = new Object();
  static final Object LOCK_B = new Object();

  public static void main(String[] args) {
    Thread t1 = new Thread(() -> {
      synchronized (LOCK_A) {
        dormir(50);
        synchronized (LOCK_B) {
          System.out.println("T1 concluiu");
        }
      }
    });

    Thread t2 = new Thread(() -> {
      synchronized (LOCK_B) {
        dormir(50);
        synchronized (LOCK_A) {
          System.out.println("T2 concluiu");
        }
      }
    });

    t1.start();
    t2.start();
  }

  static void dormir(long ms) {
    try { Thread.sleep(ms); } catch (InterruptedException e) {
Thread.currentThread().interrupt(); }
  }
}
```

- O deadlock decorre de exclusão mútua sobre os locks, manter-e-esperar (cada thread segura um lock e aguarda outro), não preempção (locks não podem ser tomados à força) e, crucialmente, espera circular, satisfazendo as quatro condições necessárias.[3]
- Correção solicitada aos alunos: impor uma ordem global e sempre adquirir LOCK_A antes de LOCK_B em todas as threads, quebrando a espera circular por hierarquia de recursos, ou reestruturar para alocação atômica/semáforo único por região crítica, removendo "hold and wait" na região problemática.[1][3]

O que entregar

- Código que reproduz o deadlock, com logs que evidenciem a ordem de aquisição e o travamento observado, acompanhado de análise relacionando com Coffman.[3]
- Implementação corrigida e explicação de como a estratégia aplicada removeu a condição de espera circular ou manter-e-esperar, relacionando com o tratamento de deadlock e a solução por hierarquia de recursos do problema dos filósofos.[1][3]

Critérios de avaliação

- Clareza conceitual ao explicar impasse e fome no Jantar dos Filósofos, bem como a condição de corrida e o papel de semáforos.[2][1]
- Correção do protocolo proposto na Parte 1 (ausência de impasse) e uso apropriado de fairness quando relevante para evitar inanição.[1]
- Demonstração reprodutível da condição de corrida e sua eliminação com Semaphore, incluindo discussão de happens-before e visibilidade.[2]
- Reprodução do deadlock, mapeamento das quatro condições e correção fundamentada por hierarquia de recursos ou outra técnica que quebre uma condição necessária.[1][3]

Referências conceituais para consulta dos alunos: descrição formal do Jantar dos Filósofos, estratégias clássicas de solução, API de java.util.concurrent.Semaphore e condições de Coffman para deadlock, incluindo prevenção pela remoção de espera circular.[2][3][1]

Toda entrega deve estar no github e os acadêmicos em um grupo no AVA

Em conjunto ao relatório, gravem um vídeo de no máximo 10 minutos explciando o trabalho. Todos devem explicar, atentar-se ao audio do vídeo, volume e qualidade da imagem. Vídeos de baixa qualidade sofrerão penalização. Adicione o link do video no github.

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