Performance em Sistemas Ciberfísicos - TDE 3

Beatriz Ceciliato Robaskievicz da Cunha - Pedro Muller Volpe | Ciência da Computação | 3U | Manhã

Parte 1 - Relatório

Resumo do problema

O *Jantar dos Filósofos* modela N filósofos sentados em círculo; cada filósofo alterna entre **pensar**, **com fome** e **comer**. Para comer, o filósofo precisa pegar dois garfos (os recursos) — o da sua esquerda e o da sua direita — que são compartilhados com os vizinhos. Se todos seguirem o protocolo ingênuo “pegar primeiro o garfo da esquerda, depois o da direita”, existe um cenário em que cada filósofo pega o garfo à sua esquerda e espera pelo da direita — ninguém consegue progredir. Esse estado caracteriza **deadlock** (impasse).

Por que o impasse surge (conexão com Coffman)

Um deadlock exige simultaneamente quatro condições (Condições de Coffman):

1. **Exclusão mútua** — cada garfo só pode ser usado por um filósofo por vez (verdadeiro).

2. **Manter-e-esperar (hold-and-wait)** — filósofos podem segurar um garfo enquanto aguardam o outro (verdadeiro no protocolo ingênuo).

3. **Não preempção** — um garfo não pode ser forçado a ser liberado por outro (verdadeiro).

4. **Espera circular** — existe um ciclo de filósofos cada um esperando por um recurso retido pelo próximo (verdadeiro no caso crítico).

No protocolo ingênuo todas as quatro condições podem ocorrer e, portanto, o sistema pode entrar em impasse.

Estratégia escolhida para evitar impasse

Vou usar a estratégia **hierarquia de recursos** (impor uma ordem global/numérica nos garfos). A ideia: atribuir índices únicos aos garfos e forçar cada filósofo a adquirir sempre **primeiro o garfo de menor índice e depois o de maior índice**. Isso **quebra a condição de espera circular** — não é possível formar um ciclo consistente de espera se todos solicitam recursos numa ordem crescente fixa.

Por que isso evita deadlock

Ao impor uma ordem total (ou consistente) sobre recursos, elimina-se a possibilidade de que A espere por B, B espere por C, ... e Z espere por A — pois as requisições seguem a ordem numérica e um filósofo nunca espera por um recurso “menor” se já possui um “maior” que outro possa estar aguardando. Logo, a **condição 4 (espera circular)** é negada, impedindo deadlock.

Pseudocódigo (hierarquia de recursos)

pseudocodigo.java

```
Dados:
  N = 5     // número de filósofos
  garfos[0..N-1]  // cada garfo é um recurso exclusivo (mutex)

Funções auxiliares:
  pensar()     // filósofo pensa por um tempo aleatório
  comer()      // filósofo come por um tempo aleatório
  adquirir(x)  // bloqueia até o garfo x ficar livre (oper. atômica)
  liberar(x)   // libera o garfo x

Para cada filósofo p (0..N-1) executar em loop:
  esquerda = p          // garfo à esquerda (entre p e (p+N-1)%N)
  direita  = (p + 1) % N
  left  = min(esquerda, direita)
  right = max(esquerda, direita)

  enquanto verdade:
    pensar()
    estado[p] <- "com fome"

    // adquirir sempre em ordem de índice: primeiro o menor, depois o maior
    adquirir(left)    // bloqueia até garfo 'left' estar livre
    adquirir(right)   // bloqueia até garfo 'right' estar livre

    estado[p] <- "comendo"
    comer()

    // liberar na ordem inversa (boa prática) — mas qualquer ordem serve
    liberar(right)
    liberar(left)
    estado[p] <- "pensando"
```

Notas sobre o pseudocódigo

- adquirir/liberar são operações atômicas sobre o recurso (por exemplo: lock/monitor/mutex).
- A escolha de liberar em ordem inversa (right antes de left) é uma prática comum para reduzir tempo de retenção, mas não é necessária para evitar deadlock — a eliminação do *circular wait* vem da ordem fixa de aquisição.
- Não há uso de timeouts ou preempção; a correção depende apenas da ordenação global.

Garantia de justiça e progresso (inanição / starvation)

Impor hierarquia resolve deadlock (bloqueio total), mas pode gerar preocupação com **inanição** se a implementação dos locks não for justa. Para mitigar:

- Use locks/semaphores com **fairness** (FIFO) quando disponíveis — por exemplo, estruturas que garantem ordem de espera.

- Alternativamente, combine a hierarquia com **backoff aleatório** e limites máximos de espera: se um filósofo espera tempo demais, ele libera e tenta novamente depois de um tempo aleatório — reduz a chance de starvation na prática.

- Em sistemas pequenos (N=5), e com locks justos, a hierarquia normalmente preserva **progresso** (algum filósofo sempre poderá comer) e evita bloqueio global.

Parte 2 - Relatório

Dinâmica do problema

Nesta parte, analisamos o comportamento de múltiplas threads acessando e modificando uma variável compartilhada. O objetivo é demonstrar como uma operação aparentemente simples, count++, pode produzir resultados incorretos quando realizada concorrentemente, caracterizando uma **condição de corrida** (*race condition*). Em seguida, empregamos um **semáforo binário** como mecanismo de exclusão mútua para garantir correção e visibilidade entre as threads.

Por que ocorre condição de corrida

A expressão count++ não é atômica. Ela equivale a três passos:

1. leitura do valor atual de count;

2. incremento;

3. escrita do novo valor na memória.

Quando várias threads executam essa sequência simultaneamente, leituras e escritas se sobrepõem, ocasionando **perda de incrementos**. Como não há qualquer forma de sincronização, o resultado final frequentemente é **menor que o valor esperado**.

Além disso:

- Não existe exclusão mútua sobre a variável,
- Não há garantias de visibilidade entre as threads,
- O compilador e a JVM podem reordenar instruções.

Esse cenário torna o comportamento **não determinístico**, assegurando que cada execução produza resultados diferentes.

Correção com semáforo binário

Para garantir exclusão mútua, utilizamos um semáforo inicializado com:

```
new Semaphore( permits: 1,  fair: true);
```

A contagem igual a 1 cria uma seção crítica exclusiva; o parâmetro true ativa o modo **justo (FIFO)**, evitando que threads esperem indefinidamente. O bloco crítico é protegido entre:

```
    sem.acquire();
    count++;
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
} finally {
    sem.release();
}
```

O semáforo estabelece uma relação de sincronização do tipo **happens-before**: qualquer thread que adquirir a permissão após uma liberação tem acesso consistente ao valor atualizado de count.

O resultado desta versão deve ser **exatamente o valor esperado**: T × M.

Discussão: trade-offs

- Sem sincronização:
    - maior throughput devido à ausência de coordenação,
    - porém resultados incorretos e comportamento imprevisível.
- Com semáforo:

- garante correção e visibilidade,
- evita perda de incrementos,
- porém reduz o desempenho, pois cada incremento é serializado.

Considerações finais

A ausência de sincronização leva a resultados incorretos devido à condição de corrida. Já o uso de semáforo garante exclusão mútua e visibilidade adequada entre threads, corrigindo o problema ao custo de maior tempo de execução.

Parte 3 - Relatório

Descrição do cenário

Nesta parte, analisamos um cenário clássico de deadlock envolvendo duas threads e dois locks distintos. Cada thread tenta adquirir dois recursos (A e B), porém em ordens opostas:

- **Thread 1**: adquire **A** → depois tenta adquirir **B**
- **Thread 2**: adquire **B** → depois tenta adquirir **A**

Se cada thread adquirir o primeiro lock com sucesso e, em seguida, ambas aguardarem mutuamente pelo segundo lock que está sendo mantido pela outra thread, o sistema entra em **deadlock**.

Relação com as condições de Coffman

O deadlock ocorre exatamente porque as quatro condições necessárias são satisfeitas simultaneamente:

1. **Exclusão mútua**
   Os locks A e B só podem ser mantidos por uma thread por vez.

2. **Manter-e-esperar (hold and wait)**
   Cada thread retém seu primeiro lock enquanto aguarda o segundo.

3. **Não preempção**
   Locks não são retirados à força; a thread só libera voluntariamente.

4. **Espera circular**
   Existe um ciclo de dependência:

    - T1 espera B (retido por T2),
    - T2 espera A (retido por T1).

Como as quatro condições estão presentes, o sistema pode ficar bloqueado indefinidamente.

Estratégia de correção: imposição de ordem global

Para evitar a espera circular, adotamos a mesma técnica aplicada ao problema dos filósofos na Parte 1: **hierarquia de recursos**.

A ideia é simples:

- Todos os recursos recebem uma **ordem fixa**.
- Todas as threads devem adquirir recursos **sempre na mesma ordem**.

Ao impor essa regra:

- elimina-se a possibilidade de criar ciclos de espera,
- impedindo assim a ocorrência de deadlock.

Como consequência, a condição 4 (espera circular) é negada, tornando o deadlock impossível.

Considerações finais

A prevenção de deadlock foi obtida eliminando-se a condição de espera circular por meio da hierarquia de recursos. Essa técnica é simples, eficiente e amplamente utilizada tanto em sistemas operacionais quanto em aplicações concorrentes que dependem de múltiplos locks simultaneamente.