TDE 3 – Performance em Sistemas Ciberfísicos

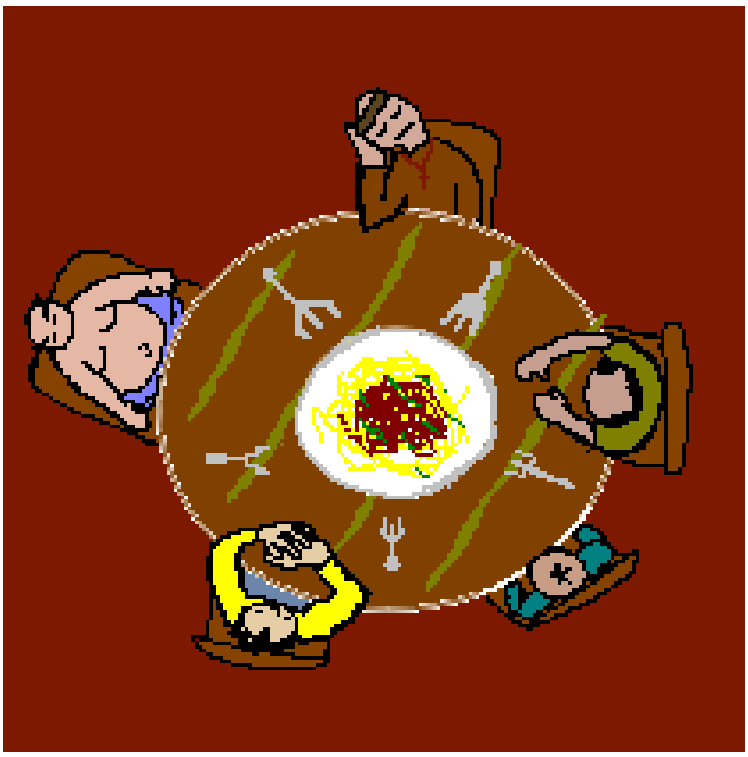
PUCPR - Professor Andrey Cabral

Aluno Davi Delfes

Parte 1 – Jantar dos Filósofos

Dinâmica do problema:

Cada filósofo precisa ter em posse dois garfos ao mesmo tempo para poder comer. Entretando, existe apenas um garfo entre cada um deles:



Portanto, se em dado momento cada filósofo pega o garfo à sua direita, ou o garfo à sua esquerda, cada um vai ter um garfo, porém como precisam de dois garfos para comer, não conseguirão fazê-lo, pois nenhum garfo está disponível e eles só soltam os garfos após comerem. Essa situação de espera para liberação do garfo, mas sem nenhum filósofo comendo configura um deadlock.

A solução de Edsger Dijkstra, a mente por trás desse problema, implementa uma solução que remove a condição de *resource holding:* um processo estar em posse de pelo menos um recuros e pedindo o uso de mais recursos que estão em posse de outros processos.

Essa solução determina que os filósofos só tentem comer se tiverem ambos os garfos adjascentes disponíveis ao mesmo tempo. Ou seja, só se os filósofos adjascentem não estiverem comendo.

Pseudocódigo:

```
int i = 0
const N = 5
enum Estado { PENSANDO, FAMINTO, COMENDO }

var estado[i..N-1] = PENSANDO

var semaforo[N] = new Semaphore(0) // um semáforo individual para cada filósofo

var lock = new Semaphore(1)   // semáforo para garantir que 2 filósofos não possam
alterar seu estado ao mesmo tempo (acesso à área crítica)

int esquerda(i): return (i + N - 1) % N //retorna indice do filósofo à esquerda
int direita(i): return (i + 1) % N //retorna indice do filósofo à diretia

void pegar_garfos(i):
    acquire(lock)
    estado[i] = FAMINTO
    testar(i)
    release(lock)
    acquire(semaforo[i])   // bloqueia filósofo até poder comer

void largar_garfos(i):
    acquire(lock)  //entra na região crítica
    estado[i] = PENSANDO
    testar(esquerda(i))   // verifica se vizinho da esquerda pode comer
    testar(direita(i))    // verifica se vizinho da direita pode comer
    release(lock)  //sai da região crítica

void testar(i):
    if (estado[i] == FAMINTO and estado[esquerda(i)] != COMENDO and
estado[direita(i)] != COMENDO)
      {
        estado[i] = COMENDO
        release(semaforo[i])   // libera filósofo i para comer
      }


void Filosofo(i):
    while true:
        pensar()
        pegar_garfos(i)
        comer()
        largar_garfos(i)
```

Parte 2 – Threads e semáforos

Resultado do teste onde threads acessam a mesma variável sem controle:

```
Esperado=2000000, Obtido=248689, Tempo=0.048s
```

Resultado do teste com controle de acesso à variável com semáforo:

```
Esperado=2000000, Obtido=2000000, Tempo=7.650s
```

Por que isso ocorre:

Uma **condição de corrida** ocorre quando duas ou mais threads acessam e modificam uma variável compartilhada **sem coordenação**.

No caso do contador, a operação `counter++` parece simples, mas na verdade envolve três passos:

1. Ler o valor atual de count.
2. Incrementar count em 1.
3. Escrever o novo valor na memória.

Porém se duas threads executam ao mesmo tempo, ambas podem ler o mesmo valor antes que a outra escreva, como no exemplo a seguir:

count = 100

Thread 1 lê count = 100

Thread 2 lê count = 100

Thread 1 calcula count++;

Thread 2 calcula count++;

Thread 1 escreve count = 101.

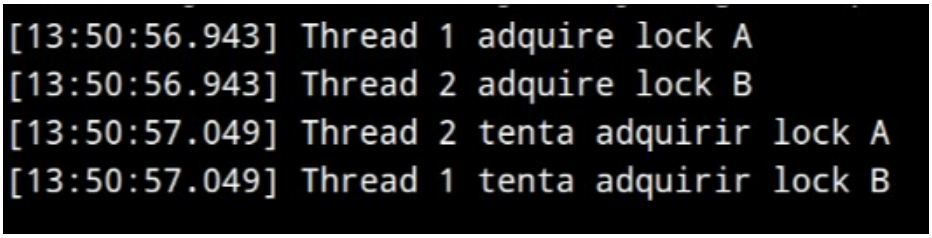
Thread 2 escreve count = 101.

Entretando o valor de count esperado após dois incrementos deveria ser 102.

Com o uso dos semáforo, cada thread adquire a única permissão disponível no semáforo. Se uma thread não tem essa permisão, ela fica bloqueada, forçando que cada thread aguarde a posse da permisão para poder executar a linha count++, responsável pela leitura, incremento e escrita do valor da variável.

Parte 3 – Deadlock

Logs do cenário de deadlock:



O travemento observado é evidenciado pelo fato da string "T[N]: obteve A/B e em seguida B/A" nunca ser impressa após a execução. Ou seja, as linhas synchronized (lockA/B) nunca são executadas, já que a Thread 1 exerce exclusão mútua quando adquire a lock A, do mesmo modo que a Thread 2 faz com a lock B.

Condições de Coffman

Exclusão Mútua: Cada lock só pode ser adquirido por uma thread por vez, garantido pelo método synchronized().

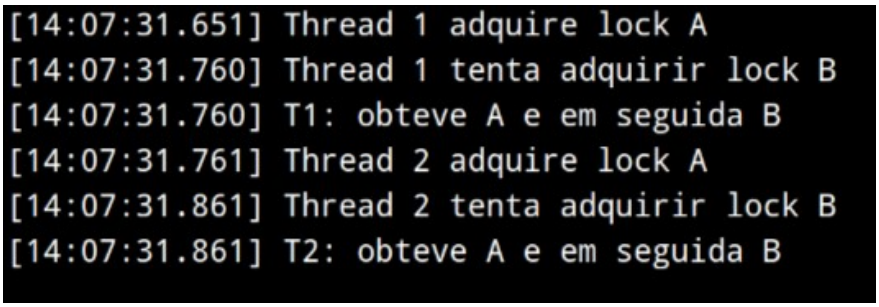
Resource holding: Ao mesmo tempo que uma thread segura um lock, aguarda pela liberação do outro que está em uso pela outra thread.

Não-preempção: Locks não podem ser tomados por força, é necessário que a thread que o controla libere voluntariamente.

Espera ciruclar: Ciclo de dependência: Thread 1 bloqueia lockA e aguarda por lockB, enquanto Thread 2 bloqueia lockB e aguarda por lockA.

Solução do deadlock com hierarquia de recursos:

Para resolução desse impasse, basta implementar uma ordem global de aquisição de recursos: ambas as threads tentam adquirir lockA e depois lockB.



Pela definição da Wikipedia:

> 4. *Circular wait:* each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, $P = \{P_1, P_2, ..., P_N\}$, such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3 and so on until P_N is waiting for a resource held by P_1.[4][9]

Ao implementar a hierarquia de recursos, essa condição é eliminada já que a o ciclo anterior onde Thread 1 bloqueia lock A enquanto aguarda lockB e Thread2 bloqueia lockB enquanto aguarda lockA não se forma: a Thread 2 aguarda que a Thread 1 libere lockA já que segue a mesma ordem de obtenção de recursos.