Pontifícia Universidade Católica do Paraná – PUCPR
Escola Politécnica – Campus Sede – Curitiba/PR
Curso Bacharelado em Ciência da Computação
Disciplina de **Performance em Sistemas Ciberfísicos**
Abílio Pedro Alcântara Mota Batista

TDE 3 – TRABALHO FINAL

Parte 1 – Filósofos

A dinâmica do problema é que o jantar dos filósofos representa cinco processos (os filósofos) que ficam alternando entre pensar, ficar com fome e comer. Para cada filósofo comer, precisa adquirir dois recursos que são compartilhados: os garfos à sua esquerda e à sua direita. Isso mostra uma questão de exclusão mútua em sistemas concorrentes, porque um garfo só pode estar com um filósofo por vez.

O impasse surge no protocolo ingênuo porque o protocolo ingênuo diz: primeiro pegue o garfo da esquerda, depois pegue o garfo da direita e, então, coma. Note que se todos os filósofos ficarem com fome na mesma hora, cada um pega o garfo da esquerda. Nenhum garfo da direita está disponível, e todos ficam esperando indefinidamente. Esse bloqueio mútuo caracteriza um deadlock.

Para que um deadlock aconteça, quatro condições devem existir ao mesmo tempo:

1. Exclusão mútua: Pelo menos um recurso deve ser usado por apenas um processo por vez. Um processo que já está usando um recurso o mantém em modo não compartilhável. Ou seja, os garfos são indivisíveis;
2. Espera e retenção: Um processo pode estar mantendo um ou mais recursos enquanto aguarda por outros recursos que estão sendo utilizados por outros processos. Ou seja, cada filósofo segura um recurso enquanto espera outro;
3. Sem preempção: Recursos não podem ser retirados à força de um processo que os detém. O processo só pode liberar o recurso voluntariamente após a sua conclusão. Ou seja, não pode tomar um garfo à força;
4. Espera circular: Existe um ciclo de processos, em que cada processo na cadeia está aguardando um recurso que está sendo mantido pelo próximo processo na cadeia. Por exemplo, o processo 1 espera por um recurso do processo 2, o processo 2 espera pelo processo 3, e o processo 3 espera pelo processo 1.

Assim, no protocolo ingênuo, nota-se todas as condições e o sistema pode acabar entrando em impasse.

Para evitar o impasse, podemos usar a hierarquia de recursos. A ideia é dar aos garfos uma ordem que é global (por exemplo, de 0 a 4) e obrigar todos os filósofos a sempre pegar primeiro o garfo que tiver o menor índice, e

depois o que tiver maior índice. Dessa forma, acabamos com a espera circular, porque nunca pode existir um ciclo de dependências se todos seguem a mesma ordem. Como citei anteriormente, para o deadlock acontecer, as quatro condições devem acontecer ao mesmo tempo. Sem a espera circular, não tem como ter deadlock.

Para o pseudocódigo, temos que N = 5 filósofos, os garfos são numerados de 0 a 4 e o garfo i fica entre o filósofo i e o filósofo (i + 1) mod N

Dados:
N = 5
Garfos 0...N-1
garfo_esq(p) = p
garfo_dir(p) = (p + 1) mod N

Para cada filósofo p:

Temos um loop:
pensar()
estado[p] <- "com fome"

left = min(garfo_esq(p), garfo_dir(p))
right = max(garfo_esq(p), garfo_dir(p))

adquirir(left)
adquirir(right)

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

liberar(right)
liberar(left)

estado[p] <- "pensando"

Assim, sempre pega primeiro o garfo de menor índice e depois o de maior, então não tem como formar um ciclo de dependência e o sistema não trava.

Parte 2 – Threads e Semáforos

A condição de corrida acontece quando duas ou mais threads acessam e atualizam um mesmo recurso ao mesmo tempo sem sincronização. No caso do contador, cada thread está executando vários incrementos, mas

a operação count++ faz tudo de uma vez só, sem que outra thread intervenha. Tem três passos: ler o valor, somar 1 e escrever de volta. Quando várias threads fazem isso na mesma hora, os passos podem acabar se intercalando e sendo perdidos, por isso que o valor final do contador fica menor do que se espera.

Para mostrar esse problema, dá para criar um contador compartilhado e lançamos T threads, cada uma fazendo M incrementos. Sem ter controle de acesso, o resultado do final normalmente é menor que TxM, mostrando a condição de corrida. Isso acontece porque múltiplas threads estão atualizando o mesmo valor ao mesmo tempo, sem exclusão mútua.

Para corrigir isso, usamos um semáforo binário, que funciona como um mecanismo de exclusão mútua. Com arquire() e release(), garantimos que apenas uma thread por vez entre na região crítica onde o contador é incrementado. Assim, nenhum incremento se perde e o valor é exatamente TxM. Ao configurar o semáforo no modo justo, temos uma ordem FIFO entre as threads, então evita-se que alguma thread seja deixada esperando.

Mas isso tem um custo: o uso do semáforo reduz o throughout, porque as threads precisam esperar sua vez pra entrar na região crítica, então o programa vai ficar lento, mas vai estar corrigido. O modo justo pode reduzir ainda mais o desempenho, mas melhora a ordem de acesso e evita espera indefinida. Por fim, o semáforo também garante uma relação de memória do tipo happens-before entre release() de uma thread e acquire() de outra, assegurando visibilidade e consistência do valor do contador. Com o semáforo binário, podemos montar o seguinte pseudocódigo:

```
count = 0
sem = Semaforo(1) <- binário

Para cada thread:
    Repetir M vezes:
        sem.acquire()
        count = count + 1
        sem.release()

Resultado vai ser TxM
```

Assim, o semáforo elimina a condição de corrida ao garantir exclusão mútua, mas tem o impacto no desempenho por causa da serialização dos acessos ao contador compartilhado.

Parte 3 – Deadlock

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

   public static void main(String[] args) {

      Thread t1 = new Thread(() -> {
         System.out.println("T1 tentando adquirir LOCK_A");
         synchronized (LOCK_A) {
            System.out.println("T1: adquiriu LOCK_A");
            dormir(50);
            System.out.println("T1 tentando adquirir LOCK_B");
            synchronized (LOCK_B) {
               System.out.println("T1 adquiriu LOCK_B");
               System.out.println("T1 concluído");
            }
         }
      });

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

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

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

Esse é o código do deadlock com os logs. Ao rodar o código, temos a seguinte saída:

T1 tentando adquirir LOCK_A
T2 tentando adquirir LOCK_B
T2 adquiriu LOCK_B
T1: adquiriu LOCK_A
T1 tentando adquirir LOCK_B
T2 tentando adquirir LOCK_A

A partir desse ponto, T1 está segurando LOCK_A e esperando LOCK_B, T2 está segurando LOCK_B e esperando LOCK_A e nenhuma delas consegue avançar. Nota-se o deadlock formado. O programa não imprime mais nada e fica travado sem lançar erro.

Analisando a relação com as condições de Coffman:

1. Exclusão Mútua: LOCK_A e LOCK_B são recursos exclusivos. Apenas uma thread por vez pode segurá-los.
2. Espera e retenção: cada thread mantém o primeiro lock e espera pelo segundo.
3. Sem preempção: locks não são tomados à força de outra thread.
4. Espera circular: T1 espera por LOCK_B que está em T2; T2 espera por LOCK_A que está em T1, aí temos um ciclo formado.

Com essas condições existindo ao mesmo tempo, temos um deadlock.

Esse é o código depois da correção:

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

   public static void main(String[] args) {

      Thread t1 = new Thread(() -> {
         System.out.println("T1: tentando adquirir LOCK_A");
         synchronized (LOCK_A) {
            System.out.println("T1: adquiriu LOCK_A");
            dormir(50);
            System.out.println("T1: tentando adquirir LOCK_B");
            synchronized (LOCK_B) {
               System.out.println("T1: adquiriu LOCK_B");
               System.out.println("T1: concluído");
            }
```

```
        }
    });

    Thread t2 = new Thread(() -> {
        System.out.println("T2: tentando adquirir LOCK_A");
        synchronized (LOCK_A) {
            System.out.println("T2: adquiriu LOCK_A");
            dormir(50);
            System.out.println("T2: tentando adquirir LOCK_B");
            synchronized (LOCK_B) {
                System.out.println("T2: adquiriu LOCK_B");
                System.out.println("T2: concluído");
            }
        }
    });

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

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

Na correção, as duas threads passam a adquirir os recursos sempre na mesma ordem fixa: primeiro LOCK_A, depois LOCK_B e isso elimina que uma thread segure LOCK_A enquanto outra segura LOCK_B. Funciona porque tiramos a condição de espera circular, que faz o deadlock acontecer. Sem essa condição, mesmo que duas threads tentem acessar os recursos ao mesmo tempo, uma delas sempre terá de aguardar, mas vai prosseguir, porque não tem um estado em que ficam esperando a outra liberar recurso.

Essa é a solução por hierarquia de recursos, assim como fizemos no problema dos filósofos anteriormente. Quando todos respeitam a mesma ordem para pegar os recursos, não podem formar ciclos de espera. Ou seja, mesmo que duas threads queiram os mesmos locks ao mesmo tempo, sempre vai ter uma delas que consegue seguir e terminar sua execução. O deadlock já não acontece mais porque não tem mais espera circular.

Pontifícia Universidade Católica do Paraná – PUCPR
Escola Politécnica – Campus Sede – Curitiba/PR
Curso Bacharelado em Ciência da Computação
Disciplina de **Performance em Sistemas Ciberfísicos**
Abílio Pedro Alcântara Mota Batista

Conclusão

Nesse trabalho, analisamos e implementamos soluções para problemas clássicos de concorrência em sistemas computacionais. Na parte 1, exploramos o jantar dos filósofos, identificando o impasse e a fome, e aplicamos a hierarquia de recursos para eliminar deadlock e garantir que todos os filósofos pudessem comer. Na parte 2, demonstramos a condição de corrida com um contador compartilhado e utilizamos um semáforo binário para garantir exclusão mútua, discutindo também fairness e a relação happens-before para assegurar visibilidade e consistência entre threads. Na parte 3, reproduzimos um deadlock com os logs, identificamos as condições de Coffman e corrigimos o problema com hierarquia de recursos, de forma que a espera circular foi eliminada.