ANOTAÇÕES JANTAR DOS FILOSOFOS:

Problemas clássicos de sincronização em S.O

Conceitos necessários para entender esse problema:

- Thread: É a menor unidade básica de execução dentro de um processo. Uma thread é uma subdivisão leve de um processo que pode ser agendada e executada de forma independente, mas compartilha o contexto do processo principal com outras threads. Threads compartilham o mesmo espaço de endereçamento e recursos (como arquivos e variáveis globais) do processo pai, mas cada uma possui eu próprio contador de programa, pilha e registradores. Threads permitem que um programa execute várias tarefas simultaneamente, aproveitando melhor processadores com múltiplos núcleos.

- Primitivas de Sincronização: São recursos de software essenciais fornecidos por uma plataforma para facilitar a interação entre threads ou processos. Comumente construídas a partir de operações de baixo nível, como operações atômicas e spinlocks, essas implementações podem se manifestar como mutexes, lock e semaphore.

- Lock: É uma estrutura de controle que restringe o acesso a um recurso compartilhado de modo que apenas uma thread ou processo de cada vez possa usá-lo. Quando uma thread adquire um lock, outras que tentarem acessá-lo devem esperar(bloquear) até que o lock seja liberado.

- Seção Crítica: Parte do código que acessa um recurso compartilhado e que não pode ser executada simultaneamente por múltiplas threads.

- Mutual Exclusion (Exclusão mútua): Garante que apenas uma thread de cada vez possa executar uma seção critica.

- Deadlock (impasse): Um conjunto de processos estará em situação de deadlock se todo processo pertencente a esse conjunto estiver a espera de um evento que

somente um outro processo desse mesmo conjunto poderá realizar. De uma forma mais simples, podemos dizer que deadlock é um termo empregado para traduzir um problema ocorrido quando um grupo de processos competem entre si.

- Starvation (inanição): Quando uma thread nunca obtém o lock porque outras sempre adquirem primeiro. De forma mais simples, ocorre inanição quando um processo nunca é executado ("morre de fome"), pois processos de prioridade maior sempre o impedem de ser executado.

- Busy Waiting (espera ativa): Técnica onde a thread espera em um loop, verificando rapidamente se o lock está disponível, consumindo CPU.

O JANTAR DOS FILOSOFOS:

1. Introdução ao Problema:

   O jantar dos filósofos é um problema fundamental de sincronização concorrente e gerenciamento de recursos em S.O's que foi proposto por Dijkstra em 1965 como um problema de sincronização. A partir de então todos os algoritmos propostos como soluções de sincronização acabaram sendo relacionados ou testados contra o problema do Jantar dos filósofos.

   Esse problema é definido da seguinte forma: Cinco filósofos estão sentados em uma mesa redonda para jantar. Cada filósofo tem um prato espaguete à sua frente. Cada prato possui um garfo para pegar o espaguete. O espaguete está muito escorregadio e, para que um filósofo consiga comer, será necessário utilizar dois garfos.

   Cada filósofo alterna entre duas tarefa: comer ou pensar. Quando um filósofo fica com fome, ele tenta pegar os garfos à sua esquerda e à sua direita; um de cada vez, independente da ordem. Caso ele consiga pegar dois garfos, ele come durante um determinado tempo e depois recoloca os garfos na mesa. Em seguida ele volta a pensar.

   Obs (analogia):

- **Processos = Filósofos**
- **Recursos compartilhados = Garfos**
- Em um sistema operacional:
- Um **processo** pode precisar de múltiplos recursos (por exemplo, memória + disco).
- A obtenção parcial desses recursos pode levar a **impasses**.

  O problema em questão é: Você é capaz de propor um algoritmo que implemente cada filósofo de modo que ele execute as tarefas de comer e pensar sem nunca ficar travado?

2. Problemas envolvidos:

Problema	Descrição
Deadlock	Todos os filósofos pegam um garfo e esperam pelo outro indefinidamente. Nenhum consegue comer.
Starvation	Um ou mais filósofos nunca conseguem os dois garfos e ficam esperando para sempre.
Race Condition	Dois filósofos tentam pegar o mesmo garfo ao mesmo tempo, resultando em comportamento imprevisível.
Inconsistência	Sem controle, filósofos podem pegar dois garfos ao mesmo tempo, violando a lógica do sistema.

3. Modelagem do Problema:

- Cada filósofo é uma **thread** ou **processo** independente.
- Cada garfo é um **recurso exclusivo**, que pode ser representado por:
  - **Semáforo binário**
  - **Mutex (mutual exclusion lock)**

CICLO DE VIDA DO FILOSOFO:

```
while (true) {
   pensar();            // estado sem uso de recursos
   pegar_garfos();      // entrada na seção crítica
   comer();             // uso dos recursos (garfos)
   devolver_garfos();   // liberação da seção crítica
```

}

4. Soluções propostas:

5.

1) Solução Hierárquica de Recursos (Resource Hierarchy):

Estratégia:

- Cada garfo recebe um número único.
- Cada filósofo pega primeiro o garfo de menor número, depois o de maior.
- Evita deadlock pois não há espera circular.

Pseudocódigo:

```
if philosopher_id < (philosopher_id + 1) % N:
        first = philosopher_id
        second = (philosopher_id + 1) % N
else:
        first = (philosopher_id + 1) % N
        second = philosopher_id
wait(mutex[first]) // pegar garfo de menor número
wait(mutex[second]) // pegar garfo de maior número

eat()

signal(mutex[first]) // soltar garfo
signal(mutex[second])
```

Explicação:

- Garante que todos os filósofos seguem a mesma regra de ordem.
- Como ninguém tenta pegar os recursos em ordem contrária, não há ciclo de espera.
- Problema possível: starvation se um filósofo for sempre ultrapassado por outros.

2) Solução do Árbitro (Semáforo Global)

Estrategia:

- Um “garçom” permite que apenas N-1 filósofos sentem à mesa ao mesmo tempo.
- Isso garante que pelo menos um filósofo sempre poderá comer.

Pseudocodigo:

```
wait(room)              // no máximo N-1 filósofos entram
wait(mutex[left_fork])
wait(mutex[right_fork])

eat()

signal(mutex[left_fork])
signal(mutex[right_fork])
signal(room)
```

Explicação:

- O semáforo room impede que todos tentem comer ao mesmo tempo.
- Evita deadlock completamente, pois sempre sobra pelo menos um garfo disponível.
- Simples e eficaz, mas centralizado (precisa do árbitro)

3) Solução Distribuída de Chandy/Misra

Estratégia:

- Cada garfo tem um dono e um estado (limpo/sujo).
- Para comer o filósofo pede garfos sujos aos vizinhos.
- O garfo só é passado se estiver sujo, e depois de usá-lo, é limpo.

Pseudocódigo (resumido):

```
for each neighbor:
        if need_fork and !have_fork:
                send_request()
on receive_request(fork):
        if fork is dirty and not eating:
```

```
                send_fork()
                mark fork as clean
before_eating():
        wait until all needed forks are held
after_eating():
        mark all held forks as dirty
```

Explicação:

- Totalmente distribuído, sem árbitro ou semáforo global.
- Cada garfo tem propriedade lógica e regras de troca baseadas em necessidade.
- Evita deadlock e starvation, mas é mais complexo de implementar.
- Muito usado como exemplo de sincronização distribuída sofisticada.

5. Conclusão:

- O Jantar dos Filósofos é um **modelo didático e prático** para entender concorrência.
- Envolve **recursos críticos de SO** como:
  1. Exclusão mútua
  2. Sincronização
  3. Escalonamento justo
- Ensina que **soluções simples podem gerar problemas graves** se não forem corretamente sincronizadas.