# Experimento Prático: O Jantar dos Filósofos – Concorrência, Sincronização e Deadlocks em Python

O experimento foi elaborado para aprofundar a compreensão sobre os desafios de sincronização e concorrência em sistemas operacionais, utilizando o problema do Jantar dos Filósofos.

Você terá a oportunidade de desenvolver e observar as consequências da falta ou da aplicação incorreta de mecanismos de sincronização, bem como as estratégias para solucionar esses problemas, utilizando a linguagem de programação Python.

---

## 1. Pré-requisitos

- **Sistema Operacional**: Lubuntu 25.04 (instalado em máquina virtual ou física).
- **Conhecimento Básico**:
  - Programação em Python 3.
  - Comandos básicos do terminal Linux.
  - Conceitos de processos e threads.

---

## 2. Conceitos Essenciais para Revisão

Antes de iniciar o experimento, é fundamental que você revisite os seguintes conceitos:

- **Processos e Threads**: Entenda as diferenças e como eles representam unidades de execução em um sistema operacional. Em Python, usaremos threads, que são mais leves que processos e compartilham o mesmo espaço de memória, tornando-as ideais para problemas de sincronização dentro de um mesmo programa.
- **Seção Crítica**: Uma porção de código que acessa recursos compartilhados e que não deve ser executada concorrentemente por mais de uma thread. É crucial garantir acesso exclusivo.
- **Condições de Corrida (Race Conditions)**: Cenários onde o resultado de uma operação compartilhada depende da ordem não determinística de execução de múltiplas threads, podendo levar a resultados incorretos ou inconsistentes.
- **Mecanismos de Sincronização em Python (`threading` module)**:
  - **Mutexes (Exclusão Mútua)** / `threading.Lock`: Mecanismos para garantir que apenas uma thread acesse um recurso compartilhado (ou execute uma seção crítica) por vez.

  - **Semáforos** / `threading.Semaphore`: Objetos de sincronização que mantêm um contador. Usados para controlar o acesso a um número limitado de recursos ou para sinalizar eventos entre threads. Podem ser úteis para coordenar fases entre threads.

- **Deadlock**: Uma situação em que duas ou mais threads ficam bloqueadas indefinidamente, cada uma esperando por um recurso que está sendo mantido por outra thread no mesmo conjunto. Nenhuma thread consegue progredir.

- **Inanição (Starvation)**: Uma situação em que uma thread é constantemente impedida de acessar os recursos necessários para sua execução, mesmo que eles se tornem disponíveis periodicamente, devido a um agendamento desfavorável ou à ação de outras threads.

---

# 3. O Problema do Jantar dos Filósofos: Contexto

Imagine cinco filósofos sentados ao redor de uma mesa redonda. Cada filósofo passa a vida alternando entre duas atividades principais: **pensar** e **comer**. No centro da mesa, há uma grande travessa de espaguete. Entre cada par de filósofos adjacentes, há exatamente um garfo. Portanto, temos cinco filósofos e cinco garfos.

Para comer o espaguete, um filósofo precisa de **dois garfos**: o que está à sua esquerda e o que está à sua direita. Um filósofo não pode comer com apenas um garfo. Após comer, ele devolve os dois garfos à mesa e volta a pensar.

Este cenário, aparentemente simples, ilustra um problema complexo de alocação de recursos e sincronização. Se todos os filósofos, motivados pela fome, decidirem simultaneamente pegar o garfo à sua esquerda, todos conseguirão. No entanto, quando tentarem pegar o garfo à sua direita, descobrirão que ele já foi pego pelo vizinho. Nesse ponto, todos os filósofos estarão segurando um garfo e esperando por outro que nunca será liberado. Essa situação de bloqueio mútuo e permanente é o que chamamos de **deadlock**.

O objetivo deste experimento é simular esse problema em Python e explorar estratégias para evitar o deadlock, permitindo que os filósofos realizem suas tarefas.

---

# 4. Cenário do Experimento

O experimento será dividido em duas fases:

- **Fase 1**: Implementação Ingênua (Com Deadlock Garantido)

- **Fase 2**: Implementação com Solução (Evitando Deadlock)

**Implementação**: Você implementará o problema do Jantar dos Filósofos em Python 3, utilizando threads do módulo `threading`.

**Passo a Passo para o Desenvolvimento**:

**Configuração do Ambiente (Lubuntu 25.04)**:

1. Abra um terminal (pressione Ctrl + Alt + T).

2. Verifique se o Python 3 está instalado:

```
python3 --version
```

3. Se não estiver instalado, instale-o:

```
sudo apt update
sudo apt install python3
```

4. Crie um diretório para o seu projeto e entre nele:

```
mkdir jantar_filosofos_python
cd jantar_filosofos_python
```

## 4.1. Fase 1: Implementação Ingênua (Com Deadlock Garantido)

**Objetivo**: Observar o comportamento do sistema quando não há uma estratégia adequada para evitar o deadlock, utilizando uma abordagem que intencionalmente força o bloqueio.

Para garantir que o deadlock ocorra de forma consistente e observável, os filósofos tentarão primeiro pegar o garfo à sua esquerda. Somente quando todos os filósofos tiverem conseguido pegar seu primeiro garfo, eles tentarão, todos ao mesmo tempo, pegar o garfo à sua direita. Essa coordenação cria a condição perfeita para a espera circular e o deadlock. Usaremos semáforos para orquestrar essa sincronização.

**Conceitos Chave para Implementar**:

- **Constantes**: Defina `NUM_FILOSOFOS` (ex: 5), e pode definir tempos mínimos/máximos para `pensar` e `comer` para adicionar variabilidade (usando `random.uniform()`).
- **Recursos Compartilhados**:
  - Uma lista de objetos `threading.Lock` para representar os `garfos`. Cada `Lock` atuará como um mutex para um garfo específico.
  - Dois objetos `threading.Semaphore`:
    - `todos_pegaram_primeiro_garfo`: Um semáforo de contagem, inicializado com `0`. Cada filósofo que conseguir pegar seu primeiro garfo o incrementará (chamando `release()`).
    - `pode_pegar_segundo_garfo`: Um semáforo de contagem, inicializado com `0`. Este será usado para tentar coordenar o momento em que os filósofos tentam pegar o segundo garfo, seguindo a lógica do pseudocódigo fornecido.
- **Função** `pensar(filosofo_id)`:
  - Imprime uma mensagem indicando que o filósofo está pensando.
  - Simula o tempo de pensamento usando `time.sleep()`.
- **Função** `comer(filosofo_id)`:
  - Imprime uma mensagem indicando que o filósofo está comendo.

  - Simula o tempo de alimentação usando `time.sleep()`.

- **Função `filosofo(filosofo_id)` (Thread do Filósofo)**: Esta será a função principal executada por cada thread (filósofo), implementando o algoritmo abaixo.

- **Bloco Principal (`if __name__ == "__main__":`)**:
  - Inicialize o gerador de números aleatórios (`random.seed(time.time())`).
  - Imprima uma mensagem de início do programa.
  - Crie a lista de `garfos` (Locks) e os semáforos.
  - Crie uma lista para armazenar as threads dos filósofos.
  - Em um laço, crie um objeto `threading.Thread` para cada filósofo, configurando-o para executar a função `filosofo` com o `filosofo_id` correspondente. Adicione a thread à lista e inicie-a (chamando `thread.start()`).
  - Após iniciar todas as threads, use um laço para chamar `thread.join()` em cada uma. Em caso de deadlock, o programa principal ficará esperando aqui indefinidamente e precisará ser interrompido manualmente (Ctrl + C).

**Algoritmo para `filosofo(filosofo_id)` (Fase 1 - Deadlock Garantido)**:

```
FUNCAO filosofo(ID, garfos, todos_pegaram_primeiro_garfo,
   pode_pegar_segundo_garfo, NUM_FILOSOFOS)
garfo_esq = ID
garfo_dir = (ID + 1) MOD NUM_FILOSOFOS

ENQUANTO VERDADEIRO FACA
PENSAR(ID) // Chama a funcao pensar

// Fase 1: Pegar o primeiro garfo (esquerda)
IMPRIMIR "Filosofo ID esta com fome e tentando pegar garfo
   garfo_esq (esquerda)"
ADQUIRIR_LOCK(garfos[garfo_esq])
IMPRIMIR "Filosofo ID pegou garfo garfo_esq (esquerda)"

// Sinaliza que pegou o primeiro garfo
LIBERAR_SEMAFORO(todos_pegaram_primeiro_garfo)

// Espera pela coordenacao para a segunda fase (conforme logica
   especifica)
// Esta parte tenta sincronizar os filosofos.
PARA I DE 1 ATE NUM_FILOSOFOS FACA
ADQUIRIR_SEMAFORO(pode_pegar_segundo_garfo)
FIM PARA
// Libera as "permissoes" de volta (para a proxima iteracao do
   filosofo, se houvesse)
PARA I DE 1 ATE NUM_FILOSOFOS FACA
LIBERAR_SEMAFORO(pode_pegar_segundo_garfo)
FIM PARA

// Fase 2: Tentar pegar o segundo garfo (direita) - AQUI O DEADLOCK
    DEVE OCORRER
IMPRIMIR "Filosofo ID tentando pegar garfo garfo_dir (direita)"
ADQUIRIR_LOCK(garfos[garfo_dir]) // Esta linha provavelmente
   bloqueara todos, causando deadlock
```

```
    IMPRIMIR "Filosofo ID pegou garfo garfo_dir (direita)"

    COMER(ID) // Chama a funcao comer

    // Libera os garfos
    IMPRIMIR "Filosofo ID liberando garfo garfo_esq (esquerda)"
    LIBERAR_LOCK(garfos[garfo_esq])
    IMPRIMIR "Filosofo ID liberando garfo garfo_dir (direita)"
    LIBERAR_LOCK(garfos[garfo_dir])
    FIM ENQUANTO
    FIM FUNCAO
```

**Instruções de Execução**:

1. Salve seu código Python como `jantar_deadlock_garantido.py`.
2. Execute-o no terminal:

```
python3 jantar_deadlock_garantido.py
```

**Análise e Discussão (Fase 1)**:

- Execute o programa. Observe o comportamento. Ele deve entrar em deadlock rapidamente, com os filósofos provavelmente parando após pegarem o garfo da esquerda e, dependendo da exata implementação da sincronização com semáforos, podem travar na tentativa de adquirir do semáforo `pode_pegar_segundo_garfo` ou na tentativa de pegar o segundo garfo. Você precisará interromper o programa manualmente (Ctrl + C).
- Reflita sobre as quatro condições de deadlock (exclusão mútua, posse e espera, não preempção, espera circular). Como o seu código, especialmente com a tentativa de coordenação por semáforos para forçar uma ação simultânea, satisfaz essas condições?

## 4.2. Fase 2: Implementação com Solução para Deadlock

**Objetivo**: Implementar uma estratégia para evitar o deadlock, garantindo que todos os filósofos consigam, eventualmente, comer.

A solução mais comum para o Jantar dos Filósofos é quebrar a condição de **Espera Circular**. Uma maneira eficaz é fazer com que um dos filósofos (ou um subconjunto deles) pegue os garfos em uma ordem diferente dos demais. No nosso caso, faremos com que o filósofo com `ID == 0` pegue o garfo à sua **direita** primeiro e, só então, o garfo à sua **esquerda**. Os outros filósofos (`ID > 0`) continuarão pegando o garfo à **esquerda** primeiro e, depois, o da **direita**. Isso quebra a simetria e impede que o ciclo de espera se forme.

**Conceitos Chave para Implementar (Modificações na Função `filosofo`)**:

- Mantenha a estrutura básica de `NUM_FILOSOFOS`, a lista de `garfos` (Locks), as funções `pensar` e `comer`, e o bloco principal da Fase 1.
- A principal mudança ocorrerá na lógica de aquisição e liberação dos garfos dentro da função `filosofo`.

- **Importante**: A coordenação complexa por semáforos (`todos_pegaram_primeiro_garfo` e `pode_pegar_segundo_garfo`) usada na Fase 1 para *forçar* o deadlock **não é mais necessária e deve ser removida** para esta fase de solução. O objetivo aqui é demonstrar a solução para o deadlock inerente ao problema, não à sincronização artificial.

**Algoritmo para `filosofo(filosofo_id)` (Fase 2 - Solução)**:

```
FUNCAO filosofo(ID, garfos, NUM_FILOSOFOS) // Semaforos de
   coordenacao da Fase 1 removidos
garfo_esq = ID
garfo_dir = (ID + 1) MOD NUM_FILOSOFOS

ENQUANTO VERDADEIRO FACA
PENSAR(ID) // Chama a funcao pensar

// SOLUCAO PARA DEADLOCK: Quebra a espera circular
SE ID É IGUAL A 0 (ou qualquer outro filosofo designado como "
   excecao")
// Filosofo "excecao" pega o garfo da DIREITA primeiro
IMPRIMIR "Filosofo ID (excecao) tentando pegar garfo garfo_dir (
   direita)"
ADQUIRIR_LOCK(garfos[garfo_dir])
IMPRIMIR "Filosofo ID (excecao) pegou garfo garfo_dir (direita)"

IMPRIMIR "Filosofo ID (excecao) tentando pegar garfo garfo_esq (
   esquerda)"
ADQUIRIR_LOCK(garfos[garfo_esq])
IMPRIMIR "Filosofo ID (excecao) pegou garfo garfo_esq (esquerda)"
SENAO (os demais filosofos)
// Demais filosofos pegam o garfo da ESQUERDA primeiro
IMPRIMIR "Filosofo ID tentando pegar garfo garfo_esq (esquerda)"
ADQUIRIR_LOCK(garfos[garfo_esq])
IMPRIMIR "Filosofo ID pegou garfo garfo_esq (esquerda)"

IMPRIMIR "Filosofo ID tentando pegar garfo garfo_dir (direita)"
ADQUIRIR_LOCK(garfos[garfo_dir])
IMPRIMIR "Filosofo ID pegou garfo garfo_dir (direita)"
FIM SE

COMER(ID) // Chama a funcao comer

// IMPORTANTE: Liberar os garfos na ORDEM INVERSA da aquisicao.
SE ID É IGUAL A 0 // Filosofo "excecao"
// Pegou: Direita, depois Esquerda. Libera: Esquerda, depois
   Direita.
IMPRIMIR "Filosofo ID (excecao) liberando garfo garfo_esq (esquerda
   )"
LIBERAR_LOCK(garfos[garfo_esq])
IMPRIMIR "Filosofo ID (excecao) liberando garfo garfo_dir (direita)
   "
LIBERAR_LOCK(garfos[garfo_dir])
SENAO // Demais filosofos
// Pegaram: Esquerda, depois Direita. Liberam: Direita, depois
   Esquerda.
IMPRIMIR "Filosofo ID liberando garfo garfo_dir (direita)"
LIBERAR_LOCK(garfos[garfo_dir])
IMPRIMIR "Filosofo ID liberando garfo garfo_esq (esquerda)"
```

```
    LIBERAR_LOCK(garfos[garfo_esq])
    FIM SE
    FIM ENQUANTO
    FIM FUNCAO
```

**Instruções de Execução**:

1. Salve seu código Python como `jantar_solucao_python.py`.
2. Execute-o no terminal:

```
python3 jantar_solucao_python.py
```

**Análise e Discussão (Fase 2)**:

- Execute o programa e observe a saída. Os filósofos devem conseguir alternar entre pensar e comer sem que o programa entre em deadlock. O programa deve continuar executando.
- Como a mudança na ordem de aquisição dos garfos pelo filósofo "exceção" (ID 0) resolve o problema do deadlock? Qual das quatro condições de deadlock foi efetivamente quebrada por essa estratégia?
- Explique a importância de liberar os recursos (garfos) na ordem inversa da aquisição. O que poderia acontecer se essa ordem não fosse respeitada?
- Discuta se esta solução garante completamente a ausência de inanição (starvation). É possível que algum filósofo demore muito mais para comer do que outros, ou até mesmo não consiga comer por longos períodos?

---

# 5. Relatório do Experimento (Para Alunos)

Ao final do experimento, você deverá apresentar um relatório detalhado contendo:

1. **Introdução**:
   - Breve descrição do problema do Jantar dos Filósofos.
   - A importância de entender a sincronização e a prevenção de deadlocks em sistemas operacionais e aplicações concorrentes.
2. **Fase 1: Implementação Ingênua (Deadlock Garantido)**
   - **Código-fonte**: Inclua o código Python completo desenvolvido para esta fase.
   - **Observações e Análise**:
     - Descreva o comportamento observado durante a execução (o surgimento do deadlock). Detalhe em que ponto o programa travou e quais mensagens foram as últimas a serem exibidas para cada filósofo.
     - Identifique e explique como cada uma das quatro condições de deadlock (exclusão mútua, posse e espera, não preempção, espera circular) se manifesta no seu código e na situação de deadlock observada.

- **Conclusão da Fase 1**: Principais aprendizados sobre a ocorrência de deadlock em um cenário com recursos compartilhados e uma estratégia de aquisição simétrica ou coordenada para falha.

3. **Fase 2: Implementação com Solução para Deadlock**

   - **Código-fonte**: Inclua o código Python completo desenvolvido para esta fase.
   - **Observações e Análise**:
     - Descreva o comportamento observado durante a execução (ausência de deadlock, filósofos comendo e pensando).
     - Explique detalhadamente a estratégia utilizada para evitar o deadlock (a quebra da simetria na aquisição dos garfos).
     - Indique qual(is) das quatro condições de deadlock foi(foram) quebrada(s) pela solução implementada e como isso ocorreu.
     - Discuta a questão da inanição (starvation) na solução apresentada. Avalie se todos os filósofos têm uma chance justa e equitativa de comer ou se algum padrão de injustiça pode emergir.
   - **Conclusão da Fase 2**: Principais aprendizados sobre a prevenção de deadlock e a importância de um design cuidadoso na alocação de recursos em sistemas concorrentes.

4. **Considerações Finais**:

   - Reflexões sobre a aplicabilidade dos conceitos de deadlock e suas soluções em problemas reais de programação concorrente (ex: acesso a bancos de dados, sistemas de arquivos, APIs de rede).
   - A relevância de um bom entendimento de processos, threads e mecanismos de sincronização para o desenvolvimento de software robusto e eficiente em Python e outras linguagens.