1. Apresentação do problema

- O problema dos Leitores e Escritores
- Explicação básica do que é o problema: trata do acesso concorrente a um recurso compartilhado e sua resolução tem a ideia de resolver possíveis problemas como leitura inconsistente ou corrompida de dados
- Dar um exemplo simples e rápido (saldo de uma conta bancaria durante uma transação, algo assim)

2. Riscos e problemas de não tratar esses casos de forma adequada

- Pode gerar Condição de Corrida (Race Condition)
    - Race condition: ocorre quando dois ou mais processos acessam e manipulam dados compartilhados ao mesmo tempo e o resultado pode mudar de acordo com a ordem de acesso de cada execução
    - Exemplo: Dois processos tentando atualizar o mesmo saldo de uma conta
        - Processo A: lê o saldo (100), adiciona 50 -> novo saldo = > 150
        - Processo B: lê o saldo (100), subtrai 30, -> novo saldo => 70
        - Se os dois fizerem isso ao mesmo tempo, pode acabar salvando **70 ou 150**, quando o certo seria **120**.
- Inanição (Starvation): processos podem nunca conseguir acessar o recurso
    - De acordo com a implementação, pode ser que ou os Leitores ou Escritores não consigam realizar suas ações devido a uma convecção de prioridade
- Deadlock: travamento mutuo entre processos, dois ou mais processos ficam **esperando indefinidamente** por recursos que nunca serão liberados, porque cada um está esperando o outro.
    - Um leitor adquire o semáforo **`mutex`** para atualizar a variável **`readcount`**, enquanto ao mesmo tempo, um escritor adquire o semáforo **`write`** para começar a escrever.
      Ambos ficam esperando que o outro libere o recurso — o leitor precisa do **`write`**, o escritor precisa do **`mutex`**.
      Resultado: nenhum dos dois consegue continuar

3. Estratégias para resolver os problemas

- Exclusão Mutua: É a **garantia de que apenas um processo por vez** pode entrar na **seção crítica**
    - Seção crítica: Trecho do código onde o processo **acessa dados compartilhados** (leitura ou escrita). Requer proteção para evitar erros de concorrência.
- Leitores podem ler simultaneamente, uma vez que vão apenas ler dados, sem modificar nenhum

- Escritores precisam ter exclusividade total, já que eles sim irão modificar os dados
- Uso de semáforos para garantir os tópicos acima

- Três possíveis abordagens para o problema
  - Prioridades para Leitores
    - Leitores entram se não há escritor escrevendo
    - Pode causar inanição de escritores
  - Prioridades para Escritores
    - Um escritor em espera impede novos leitores
    - Evita inanição de escritores, mas pode causar inanição de leitores
  - Solução Justa (sem prioridade)
    - Balanceamento entre leitores e escritores
    - Evita inanição com controle de ordem de chegada ou filas

4. Implementação de código

- Mostrar as 3 implementações com semáforo, basta explicar a logica na primeira vez, nos outros dois acho que o máximo que muda são as prioridades, mas o uso do semáforo será bem parecido

5. Considerações

- É só uma abstração para mostrar o problema de sincronização
- Sistemas reais usam variações mais complexas

IMPLEMENTAÇÕES COM SEMÁFOROS

PRIORIDADE PARA LEITORES

```
semaphore mutex = 1;
semaphore write = 1;
int readcount = 0;

processo Leitor() {
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(write);  // primeiro leitor bloqueia escritores
    signal(mutex);

    // Seção crítica de leitura

    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(write);  // último leitor libera escritores
```

```
    signal(mutex);
}

processo Escritor() {
    wait(write);
    // Seção crítica de escrita
    signal(write);
}
```

PRIORIDADE PARA ESCRITORES

```
semaphore mutex = 1;
semaphore write = 1;
semaphore readTry = 1;
int readcount = 0;

processo Leitor() {
    wait(readTry);       // espera se houver escritores na fila
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(write);     // primeiro leitor bloqueia escrita
    signal(mutex);
    signal(readTry);

    // Seção crítica de leitura

    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(write);   // último libera escrita
    signal(mutex);
}

processo Escritor() {
    wait(readTry);       // bloqueia novos leitores
    wait(write);
    // Seção crítica de escrita
    signal(write);
    signal(readTry);     // libera leitores novamente
}
```

SOLUÇÃO JUSTA (SEM PRIORIDADE)

```
semaphore queue = 1;     // controla a ordem de chegada
semaphore mutex = 1;
semaphore write = 1;
int readcount = 0;

processo Leitor() {
    wait(queue);         // garante ordem de chegada
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(write);
    signal(mutex);
    signal(queue);

    // Seção crítica de leitura

    wait(mutex);
```

```
    readcount--;
    if (readcount == 0)
        signal(write);
    signal(mutex);
}

processo Escritor() {
    wait(queue);          // garante ordem de chegada
    wait(write);
    // Seção crítica de escrita
    signal(write);
    signal(queue);
}
```