

O Problema do Impasse em Sistemas Operacionais

Deadlock

Pedroso

2025

O que é o Impasse?

- Ocorre quando dois ou mais processos estão bloqueados, esperando uns pelos outros, e nunca conseguirão prosseguir.
- Cada processo no conjunto aguarda por um recurso que está sendo mantido por outro processo do mesmo conjunto.
- Situação de impasse total. Nenhum dos processos consegue avançar.

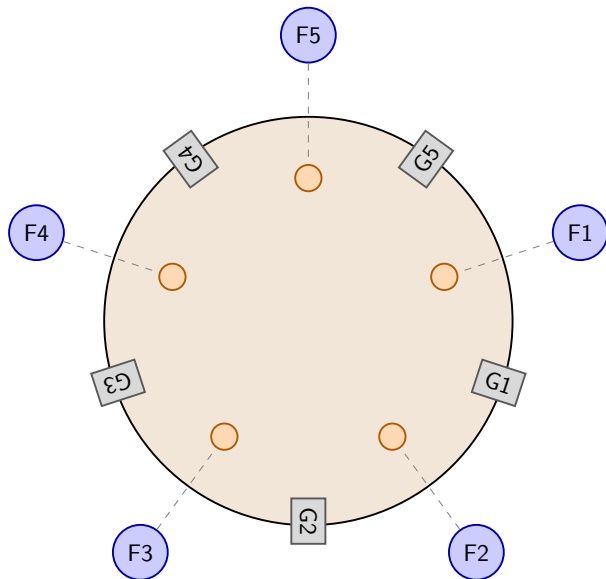
Condições para Ocorrência do Impasse

- O deadlock só pode ocorrer se as quatro condições de Coffman forem satisfeitas simultaneamente:
- **1. Exclusão Mútua:** Pelo menos um recurso deve ser não-compartilhável. Se um processo o detém, nenhum outro pode usá-lo.
- **2. Posse e Espera (Hold and Wait):** Um processo deve estar segurando pelo menos um recurso e esperando por outros recursos que estão sendo detidos por outros processos.
- **3. Não-Preempção:** Um recurso não pode ser retirado de um processo à força. Ele deve ser liberado voluntariamente pelo processo que o detém.
- **4. Espera Circular (Circular Wait):** Existe um conjunto de processos $\{P_0, P_1, \dots, P_n\}$ tal que P_0 espera por um recurso de P_1 , P_1 por um de P_2 , ..., P_{n-1} por um de P_n , e P_n por um de P_0 .

O Jantar dos Filósofos

- Um dos exemplos mais famosos de deadlock.
- **Cenário:** Cinco filósofos estão sentados ao redor de uma mesa redonda.
- Entre cada par de filósofos há um garfo.
- Para comer, um filósofo precisa de dois garfos: o da sua esquerda e o da sua direita.
- Ações:
 - ➊ Um filósofo pensa.
 - ➋ Ele fica com fome.
 - ➌ Ele pega o garfo da esquerda.
 - ➍ Ele pega o garfo da direita.
 - ➎ Ele come.
 - ➏ Ele larga os dois garfos.
 - ➐ Ele volta a pensar.

O Jantar dos Filósofos



O Jantar dos Filósofos

```
1: Thread Filosofo( $F_i$ )
2: while verdadeiro do
3:   Pensa( $F_i$ )
4:   Pega_Garfo( $G_i$ ) // garfo à esquerda
5:   Pega_Garfo( $G_{(i+1) \bmod N}$ ) // garfo à direita
6:   Come( $F_i$ )
7:   Devolve_Garfo( $G_i$ )
8:   Devolve_Garfo( $G_{(i+1) \bmod N}$ )
9: end while
```

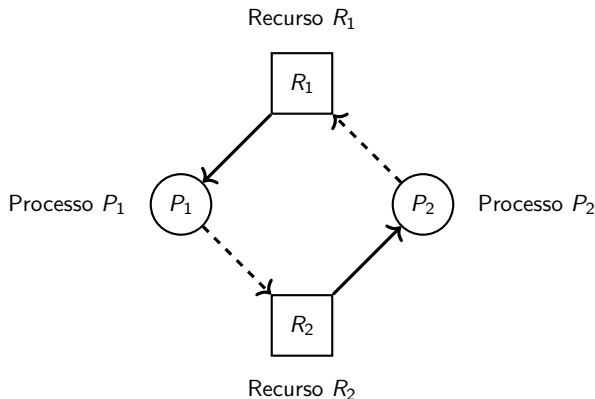
O Deadlock no Jantar dos Filósofos

- O impasse ocorre se todos os filósofos, ao mesmo tempo, pegarem o garfo da sua esquerda.
- Todos estarão segurando um garfo e esperando pelo garfo da direita, que está nas mãos do vizinho.
- Nenhum deles consegue o segundo garfo. Todos estão bloqueados indefinidamente.
- Exemplo perfeito das quatro condições de Coffman:
 - ✓ **Exclusão Mútua:** Cada garfo só pode ser usado por um filósofo por vez.
 - ✓ **Posse e Espera:** Cada filósofo detém um garfo e espera por outro.
 - ✓ **Não-Preempção:** Os garfos não podem ser retirados à força.
 - ✓ **Espera Circular:** O filósofo 1 espera por P2, P2 espera por P3, P3 espera por P4, P4 espera por P5, e P5 espera por P1.

Grafo de Alocação de Recursos

- Uma ferramenta gráfica para visualizar a alocação de recursos e a espera de processos.
- **Nós:**
 - ✓ Círculos representam **processos** ($P_1, P_2, \dots$).
 - ✓ Retângulos representam **recursos** ($R_1, R_2, \dots$).
- **Arestas:**
 - ✓ Aresta de **requisição**: $P_i \rightarrow R_j$ (Processo P_i está esperando pelo recurso R_j).
 - ✓ Aresta de **alocação**: $R_j \rightarrow P_i$ (Recurso R_j foi alocado ao processo P_i).

Exemplo de Grafo de Deadlock



- **Aresta de alocação:** Recurso alocado a processo.
- **Aresta de requisição (tracejada):** Processo espera por recurso.
- A presença de um *ciclo* no grafo é uma condição **suficiente** para o deadlock se cada recurso tiver apenas uma instância.

Estratégias de Lida com o Deadlock

- O gerenciamento de deadlock pode ser abordado de três maneiras:
- **1. Prevenção:** Projetar o sistema para que o deadlock nunca ocorra, garantindo que pelo menos uma das quatro condições de Coffman nunca seja satisfeita.
- **2. Evitação:** Permite que as quatro condições ocorram, mas faz uma alocação cuidadosa de recursos para evitar o estado de deadlock.
- **3. Detecção e Recuperação:** Permite que o deadlock ocorra, detecta quando ele acontece e age para recuperá-lo.
- **Ignorância:** A abordagem mais comum em sistemas operacionais como Windows e UNIX, onde se assume que o deadlock raramente ocorre e a penalidade por preveni-lo é muito alta.

1. Prevenção do Deadlock

- Forçar a violação de uma ou mais das condições de Coffman.
- **Exclusão Mútua:** Torne os recursos compartilháveis, se possível (raramente viável para todos os recursos).
- **Posse e Espera:**
 - ✓ Exigir que um processo solicite todos os seus recursos antes de iniciar a execução.
 - ✓ Exigir que um processo libere todos os recursos que detém antes de solicitar novos.
- **Não-Preempção:** Se um processo que detém alguns recursos solicita outro e é negado, ele deve liberar todos os recursos que detém.
- **Espera Circular:** Impor uma ordem total para a solicitação de recursos.

2. Evitação do Deadlock

- Abordagem mais dinâmica. O sistema precisa de informações adicionais sobre os recursos que os processos podem solicitar.
- **Algoritmo do Banqueiro (Banker's Algorithm):**
- Mantém o sistema em um **estado seguro**.
- Um estado é considerado seguro se houver uma sequência de execução de processos que permita que todos eles completem sua tarefa, mesmo que eles solicitem o máximo de recursos.
- Sempre que um processo solicita recursos, o algoritmo verifica se a alocação deixará o sistema em um estado seguro. Se não, o processo espera.

Exemplo Simplificado do Algoritmo do Banqueiro

- **Cenário:** Um sistema com **1 tipo de recurso** e **4 processos** (P_0, P_1, P_2, P_3).
- **Total de Recursos:** 10 instâncias.
- **Recursos Disponíveis:** 3 instâncias.
- **Situação Atual:**

Processo	Alocados	Máximo Necessário
P_0	0	7
P_1	3	5
P_2	2	6
P_3	2	4

- **Pergunta:** O estado atual é **seguro**?
- **Passo 1:** Os recursos disponíveis (3) são suficientes para atender algum processo?
- Sim! O processo P_1 ou P_3 pode ser atendido, pois sua necessidade é ≤ 3 .
- Neste caso, escolher P_1 para completar primeiro.

- **Sequência de execução:** $\langle P_1, P_3, P_0, P_2 \rangle$
- **Inicialmente:** Disponível = 3
- **1. Atender P_1 :**
 - ✓ P_1 precisa de 2 recursos ($2 \leq 3$).
 - ✓ P_1 finaliza e libera seus 3 recursos alocados.
 - ✓ **Disponível** = 3 (inicial) + 3 (liberado por P_1) = 6.
- **2. Atender P_3 :**
 - ✓ P_3 precisa de 2 recursos ($2 \leq 6$).
 - ✓ P_3 finaliza e libera seus 2 recursos alocados.
 - ✓ **Disponível** = 6 (anterior) + 2 (liberado por P_3) = 8.

Sequência Segura (Continuação)

➤ 3. Atender P_0 :

- ✓ P_0 precisa de 7 recursos ($7 \leq 8$).
- ✓ P_0 finaliza e libera seus 0 recursos alocados.
- ✓ **Disponível** = 8 (anterior) + 0 (liberado por P_0) = 8.

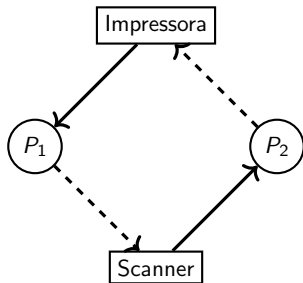
➤ 4. Atender P_2 :

- ✓ P_2 precisa de 4 recursos ($4 \leq 8$).
- ✓ P_2 finaliza e libera seus 2 recursos alocados.
- ✓ **Disponível** = 8 (anterior) + 2 (liberado por P_2) = 10.

Conclusão: Como uma sequência de execução segura foi encontrada, o estado atual é seguro.

O Problema da Impressora sem Spool

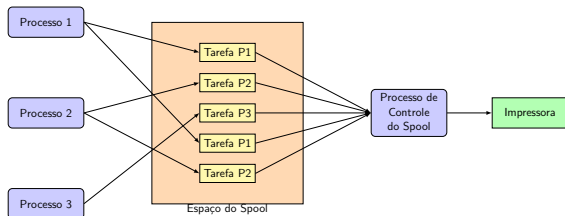
- Considere um sistema com dois processos, P_1 e P_2 , e dois recursos: uma **impressora** e um **scanner**.
- **Cenário de Impasse:**
 - ➊ P_1 solicita e obtém a **impressora**.
 - ➋ P_2 solicita e obtém o **scanner**.
 - ➌ P_1 agora solicita o **scanner** (que está com P_2).
 - ➍ P_2 agora solicita a **impressora** (que está com P_1).
- Ambos os processos estão aguardando um recurso que o outro detém.



- **Resultado:** Um **ciclo** fechado no grafo de alocação de recursos.

O que é um Spool?

- **Spool** (Simultaneous Peripheral Operations On-Line) é uma técnica de gerenciamento de dados em sistemas computacionais.
- Ele utiliza um **buffer** (normalmente no disco rígido) para armazenar dados de um dispositivo lento, como uma impressora.
- O sistema operacional pode liberar o processo rapidamente, permitindo que ele continue sua execução, enquanto o spool se encarrega de transferir os dados para o dispositivo em seu próprio tempo.



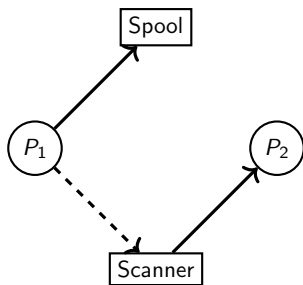
A Solução com Spool

- A chave está em **transformar um recurso não-compartilhável em um recurso compartilhável** (virtualmente).
- O processo não mais solicita a impressora física diretamente. Ele insere as tarefas de impressão no **spool de impressão**.
- Como o spool pode aceitar múltiplas tarefas de impressão ao mesmo tempo, a condição de **Exclusão Mútua** é violada.

O Cenário Modificado com Spool

➤ Novo Cenário:

- 1 P_1 envia seu trabalho para o **spool de impressão**. O processo P_1 é **liberado**.
- 2 P_2 solicita e obtém o **scanner**.
- 3 P_1 não está mais esperando por recursos da impressora e pode solicitar outros, como o scanner.
- 4 **Não há mais espera circular**, pois o recurso "Impressora" agora é o spool, que pode ser acessado por múltiplos processos.



- **Resultado:** O ciclo foi quebrado, pois o recurso da impressora agora é virtualmente compartilhável.

Exemplos de Sistemas que Implementam Spooling

- O conceito de spooling é fundamental para o gerenciamento de dispositivos de E/S lentos em praticamente todos os sistemas operacionais modernos.
- Ele permite que a CPU e outros processos não fiquem bloqueados, esperando por operações lentas.
- **Sistemas de Impressão (Print Spoolers):**
 - ✓ Quando um processo envia um documento para impressão, o sistema operacional o coloca em uma **fila de spool** no disco rígido.
 - ✓ Um processo separado (o spooler) gerencia essa fila, enviando os trabalhos para a impressora na ordem correta, sem bloquear o processo original.
 - ✓ **Exemplos:**
 - ✓ **Windows:** O serviço "Spooler de Impressão".
 - ✓ **Linux/UNIX:** Os subsistemas CUPS (Common Unix Printing System) e LPD (Line Printer Daemon).

► Outras Aplicações de Spooling:

- ✓ A técnica é utilizada em qualquer cenário onde a velocidade da CPU e a de um dispositivo de E/S são incompatíveis.
- ✓ **E-mail:** Servidores de e-mail (como o Exim ou Postfix) utilizam filas de spool para armazenar e-mails que aguardam envio, especialmente se o servidor de destino estiver indisponível temporariamente.
- ✓ **Processamento em Lote (Batch Processing):** Sistemas legados usavam spools para ler cartões perfurados ou fitas magnéticas em um ritmo constante, permitindo que a CPU processasse os trabalhos de forma eficiente em lotes.

- A técnica de **spooling** é uma forma de **prevenção de deadlock**.
- Ela ataca a condição de **Exclusão Mútua** de Coffman.
- Ao invés de um processo monopolizar o dispositivo físico, ele monopoliza um espaço de disco (o spool).
- Isso transforma o recurso em um **recurso compartilhável virtualmente**, permitindo que múltiplos processos usem o "serviço" de impressão simultaneamente.
- É uma solução elegante e amplamente utilizada para um problema comum em sistemas operacionais.

3. Detecção e Recuperação

- Permite que o deadlock ocorra. O sistema então o detecta e se recupera.
- **Detecção:**
- Usa o Grafo de Alocação de Recursos para procurar por ciclos. O algoritmo é executado periodicamente.
- Se um ciclo é encontrado, um deadlock existe.
- **Recuperação:**
- **Encerramento de processo:** Encerra um ou mais processos no ciclo de deadlock. É a abordagem mais simples, mas pode causar perdas de trabalho.
- **Preempção de recurso:** Retira um ou mais recursos de um processo e os aloca para outros, quebrando o ciclo. O processo afetado precisa reverter para um estado anterior.

O Algoritmo do Avestruz (Ostrich Algorithm)

- Esta é uma *não-solução* para o problema do deadlock.
- O nome vem da crença popular de que avestruzes enterram a cabeça na areia para ignorar o perigo.
- A ideia é **ignorar o problema do deadlock** e fingir que ele não existe.
- **Filosofia:**
 - ✓ Se a probabilidade de um impasse ocorrer é **muito baixa** e o custo de evitá-lo ou recuperá-lo é **muito alto**, a melhor estratégia é simplesmente ignorá-lo.
 - ✓ A suposição é que a ocorrência de um deadlock é tão rara que o custo de gerenciá-lo seria maior do que a perda de desempenho ou o tempo de inatividade que ele causaria.

O Algoritmo do Avestruz (Ostrich Algorithm)

► Vantagens:

- ✓ Simplicidade e alta performance.
- ✓ Não há sobrecarga de gerenciamento de recursos.

► Desvantagens:

- ✓ Se um impasse ocorrer, o sistema pode travar ou se tornar imprevisível.
- ✓ Pode exigir intervenção manual para reiniciar o sistema ou os processos afetados.

► Sistemas que o Utilizam:

- ✓ A maioria dos sistemas operacionais de propósito geral, como Windows e Linux, utiliza essa filosofia.
- ✓ Para recursos de sistema padrão, o gerenciamento de deadlock é geralmente ignorado.
- ✓ O programador de software é, em grande parte, responsável por evitar impasses no seu próprio código (por exemplo, usando mutexes e semáforos de forma correta).

Recuperação por Rollback

- É uma estratégia de **detecção e recuperação** de deadlock.
- O sistema **permite que o impasse ocorra**.
- Quando um impasse é detectado, ele **reverte o estado de um ou mais processos** para um ponto anterior, quebrando o ciclo de dependência.
- O processo é reiniciado a partir desse ponto, com a esperança de que uma nova ordem de alocação de recursos evite o problema.
- **Processo de Recuperação:**
 - ➊ **Detecção:** Um algoritmo de detecção (ex: busca por ciclos no grafo de alocação) identifica o impasse.
 - ➋ **Escolha da Vítima:** O sistema escolhe um processo para ser o sacrificado. Critérios incluem menor tempo de execução, menos recursos alocados, etc.
 - ➌ **Rollback:** O estado da vítima é desfeito, voltando para um **ponto de verificação** (checkpoint) anterior.
 - ➍ **Reinício:** O processo é reiniciado a partir do checkpoint.

Por que a Implementação é Difícil?

- A técnica de rollback não é trivial e apresenta grandes desafios.
- **Complexidade de Manter Checkpoints:**
 - ✓ O sistema deve salvar o estado completo de um processo (memória, registradores, estado de E/S, recursos, etc.).
 - ✓ Isso exige um gerenciamento de dados complexo e consome bastante espaço de armazenamento.
- **Sobrecarga (Overhead):**
 - ✓ A criação e o salvamento de pontos de verificação periódicos geram uma sobrecarga de processamento.
 - ✓ Isso pode impactar negativamente o desempenho do sistema, mesmo quando não há impasses.

Por que a Implementação é Difícil?

➤ **Seleção da Vítima:**

- ✓ A escolha de qual processo reverter é crítica e pode ter um impacto significativo no desempenho.
- ✓ Escolher a vítima errada pode levar a novos impasses ou desperdiçar muito tempo de processamento.

➤ **Risco de Reincidência:**

- ✓ Não há garantia de que o processo, ao ser reiniciado, não irá entrar em deadlock novamente na mesma sequência.
- ✓ A natureza estocástica de muitas alocações de recursos torna este risco real.

Onde o rollback é implementado?

- Sistemas comerciais aplicavam rollback.
- **NEC ACOS-4:**
 - ✓ Mainframe japonês.
 - ✓ Implementava prevenção e recuperação de deadlocks via rollback de processos.
- Hitachi mainframes e Fujitsu BS2000: rollback de processos ou transações.
- IBM OS/360 MVT e VS (anos 70-80) implementavam rollback parcial de jobs em batch para recuperação de deadlocks ou erros de I/O em mainframes.⁴
- DEC VAX/VMS – oferecia rollback de transações de subsistemas de banco de dados integrados (RMS).

Atualmente onde o roolback é implementado?

- A técnica é mais adequada para sistemas onde a consistência dos dados é mais importante do que a sobrecarga gerada.
- É raro ver essa implementação em sistemas operacionais de propósito geral (como Windows ou Linux) para gerenciar processos comuns.
- **Sistemas de Gerenciamento de Banco de Dados (SGBD):**
 - ✓ Esta é a principal área de aplicação do rollback.
 - ✓ Praticamente todos os bancos de dados relacionais modernos implementam rollback de transações como parte da propriedade ACID (Atomicidade, Consistência, Isolamento, Durabilidade). O rollback é usado para desfazer transações incompletas ou que falharam, o que também evita deadlocks em muitos casos.
 - ✓ As transações de banco de dados são frequentemente salvas em checkpoints. Se uma transação entra em deadlock ou falha, ela pode ser desfeita (rollback) para o estado inicial, garantindo a integridade dos dados.
 - ✓ **Exemplos:** Oracle, PostgreSQL, Microsoft SQL Server.