Comunicação entre Processos

LICENCIATURA EM COMPUTAÇÃO

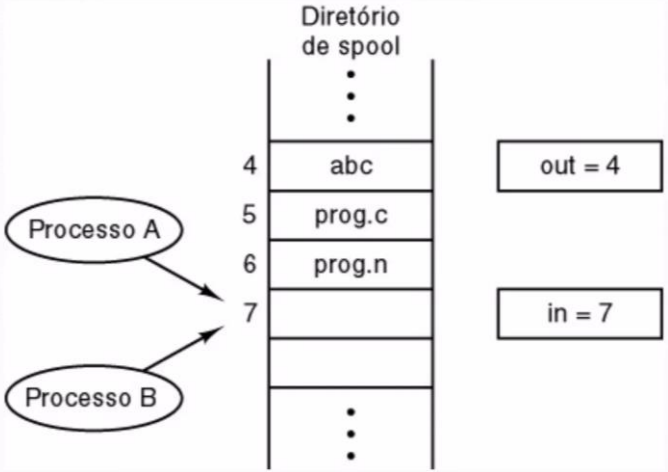
Comunicação entre Processos

- **Frequentemente, processos precisam se comunicar com outros processos.**
  - Ex.: a saída do primeiro processo tem de ser passada para um segundo e, assim por diante.
  - De preferência de uma **maneira bem estruturada** sem usar interrupções.
- **Há 3 tópicos que devem ser abordados:**
  - Como um processo passa informações para outro;
  - Como garantir que dois ou mais processos não invadam uns aos outros quando envolvidos em atividades críticas;
  - Sequência adequada quando existirem dependências.
    - Ex.: se o processo *A* produz dados e o processo *B* os imprime, *B* tem de esperar até que *A* tenha produzido alguns dados antes de começar a imprimir.

Os mesmos problemas e soluções se aplicam aos *threads*.

Condições de Disputa (Corrida)

- Situações nas quais dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende das informações de quem e quando executa precisamente.



Dois processos querem ter acesso simultaneamente à memória compartilhada

Regiões Críticas

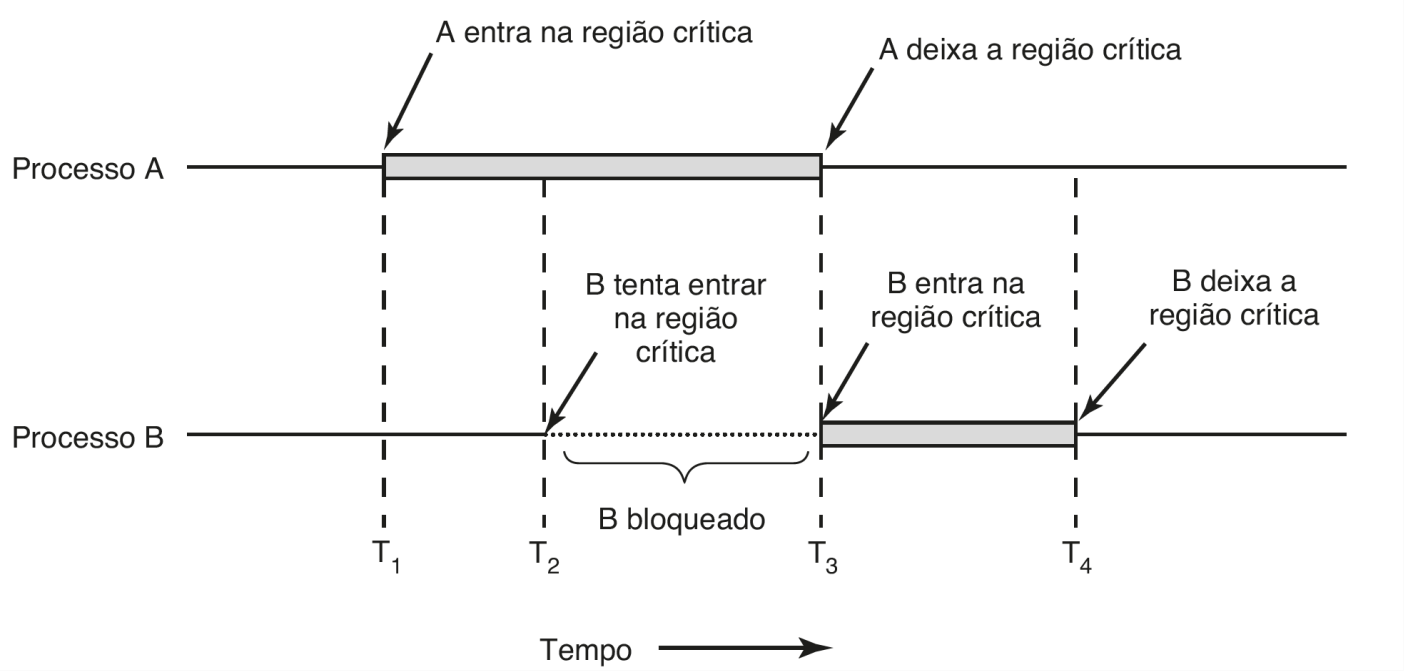
- **Como evitar as condições de corrida?**
  - A chave para **evitar problemas** em situações envolvendo memória compartilhada, arquivos compartilhados e tudo o mais compartilhado é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo.
  - **Exclusão mútua (exclusividade de acesso):** maneira de se certificar de que se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar a mesma coisa.
- Durante parte do tempo, um processo está ocupado realizando computações internas e outras coisas que não levam a condições de corrida.
- No entanto, às vezes um processo tem de acessar uma memória compartilhada ou arquivos, ou realizar outras tarefas críticas que podem levar a corridas.
  - Essa parte do programa é chamada de **região crítica** ou **seção crítica**.
  - Garantir que dois processos NÃO estejam em suas regiões críticas ao mesmo tempo.
  - Serialização.

Regiões Críticas

- Embora essa exigência evite as condições de corrida, ela não é suficiente para garantir que **processos em paralelo cooperem de modo correto e eficiente usando dados compartilhados.**
- **Necessário que quatro condições** se mantenham para chegar a uma **boa solução**:
  - Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas.
  - Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs.
  - Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo.
  - Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

Exclusão mútua usando regiões críticas

- Comportamento esperado:



Região Crítica

- Cabe ao **programador**:
  - Detectar as regiões críticas no seu código e
  - Implementar as rotinas de proteção e exclusão mútua.
- Reconhecer a parte do código que pode trazer problemas:
  - **Não é uma tarefa fácil**!
  - Exige muitos testes e tempo.

Exclusão Mútua com Espera Ociosa

- Desabilitando Interrupções.
- Variáveis de Impedimento.
- Alternância Obrigatória.
- Solução de Peterson.
- A Instrução TSL.

Desabilitando Interrupções

- **É a solução mais simples: cada processo desabilita todas as interrupções logo depois de entrar na região crítica e reabilita imediatamente antes de sair dela;**
  - Com as interrupções desligadas, a CPU não será chaveada para outro processo.
- **Não é interessante porque não é prudente dar aos processos dos usuários o poder de desligar interrupções;**
  - E se um deles desligasse uma interrupção e nunca mais a ligasse de volta? Isso poderia ser o fim do sistema.
  - Se o sistema é um multiprocessador, desabilitar interrupções afeta somente a CPU que executou a instrução *disable*. As outras continuarão executando e podem acessar a memória compartilhada.
- **É uma técnica bastante útil dentro do próprio SO, mas inadequada como um mecanismo geral de exclusão mútua para processos de usuário.**

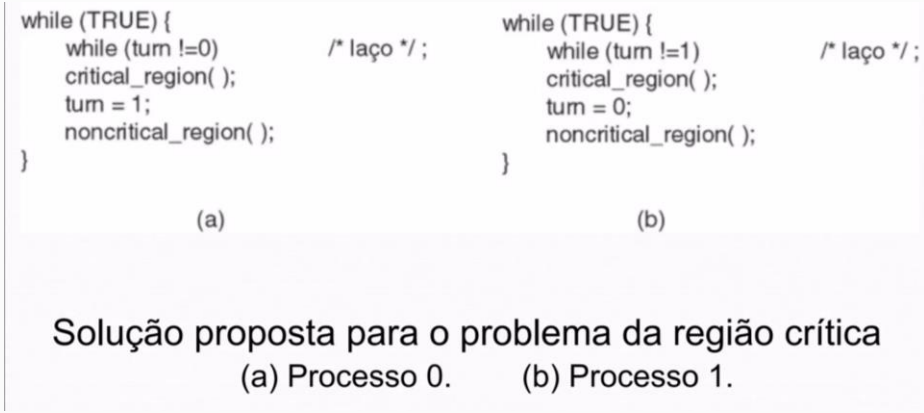
Variáveis de Impedimento

- **É uma solução de *software*;**
- **Há uma única variável compartilhada entre os dois processos (*lock* - trava), contendo o valor 0;**
- **Quando um processo quer entrar na sua região crítica, ele testa a trava:**
  - Se *lock* for 0, o processo altera essa variável para 1 e entra na região;
  - Se *lock* estiver com valor 1, o processo simplesmente aguardará até que ela se torne 0;
- **Só que essa técnica possui uma falha como diretório de *spool*.**
  - Suponha que um processo lê a trava e vê que ela é 0. Antes que ele possa configurar a trava para 1, outro processo está escalonado, executa e configura a trava para 1. Quando o primeiro processo executa de novo, ele também configurará a trava para 1, e dois processos estarão nas suas regiões críticas ao mesmo tempo.

Alternância Obrigatória

- **Contém uma variável inteira (*turn*), inicialmente com 0, que serve para controlar a vez de quem entra na região crítica e examinar ou atualizar a memória compartilhada.**
  - Inicialmente, o processo 0 inspeciona *turn*, descobre que ele é 0 e entra na sua região crítica. O processo 1 também encontra lá o valor 0 e, portanto, espera em um laço fechado testando continuamente *turn* para ver quando ele vira 1.
  - Quando o processo 0 deixa a região crítica, ele configura *turn* para 1, a fim de permitir que o processo 1 entre em sua região crítica.
- Testar continuamente uma variável até que algum valor apareça é chamado de **espera ocupada**. Em geral ela deve ser evitada, já que desperdiça tempo da CPU. Apenas quando há uma expectativa razoável de que a espera será curta, a espera ocupada é usada.
- Uma trava que usa a espera ocupada é chamada de **trava giratória** (*spin lock*).

Alternância Obrigatória



```
while (TRUE) {
    while (turn !=0)          /* laço */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn !=1)          /* laço */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Solução proposta para o problema da região crítica
(a) Processo 0. (b) Processo 1.

- **Alternar a vez não é uma boa ideia quando um dos processos for muito mais lento que o outro.**
  - O processo pode será bloqueado por um outro processo que não está na sua região crítica.
  - Na realidade, essa solução exige que os dois processos alternem-se estritamente na entrada em suas regiões críticas.

Solução de Peterson

- Antes de usar as **variáveis compartilhadas** (entrar na região crítica), cada processo chama *enter_region* com seu **próprio número de processo**, 0 ou 1, como parâmetro. Essa chamada fará que ele espere, se necessário, até que seja seguro entrar.
- Após haver terminado com as variáveis compartilhadas, o processo chama *leave_region* para indicar que ele terminou e para permitir que outros processos entrem, se assim desejarem.

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* número de processos */

int turn;                           /* de quem é a vez? */
int interested[N];                  /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process);     /* processo é 0 ou 1 */
{
     int other;                     /* número de outro processo */

     other = 1 – process;           /* o oposto do processo */
     interested[process] = TRUE;    /* mostra que você está interessado */
     turn = process;                /* altera o valor de turn */
     while (turn == process && interested[other] == TRUE) /* comando nulo */;
}

void leave_region(int process)      /* processo: quem está saindo */
{
     interested[process] = FALSE;   /* indica a saída da região crítica */
}
```

Solução de Peterson

- **Resumindo...**

- Inicialmente, nenhum processo está na sua região crítica.
- Agora o processo 0 chama *enter_region* **e** indica o seu interesse alterando o valor de seu elemento de arranjo e alterando *turn* para 0.
- Como o processo 1 não está interessado, *enter_region* retorna imediatamente.
- Se o processo 1 fizer agora uma chamada para *enter_region*, ele esperará ali até que *interested*[0] mude para *FALSE*, um evento que acontece apenas quando o processo 0 chamar *leave_region* para deixar a região crítica.

Instrução TSL

- Requer um pequeno auxílio do *hardware*.
- Computadores projetados com múltiplos processadores tem a instrução TSL RX, LOCK.
- TSL (*Test and Set Lock*) - teste e atualize a variável de impedimento: ele lê o conteúdo da palavra *lock* da memória para o registrador e então armazena um valor diferente de zero no endereço de memória *lock*.
- As operações de leitura e armazenamento de uma palavra são seguramente indivisíveis: nenhum outro processador pode acessar a palavra na memória até que a instrução tenha terminado.
- A CPU executando a instrução TSL impede o acesso ao barramento de memória para proibir que outras CPUs acessem a memória até ela terminar.

Instrução TSL

- **Entrando e saindo de uma região crítica:**
  - Para usar a instrução TSL, usa-se uma variável compartilhada, *lock*, para coordenar o acesso à memória compartilhada. Quando *lock* está em 0, qualquer processo pode configurá-lo para 1 usando a instrução TSL e, então, ler ou escrever a memória compartilhada. Quando terminado, o processo configura *lock* de volta para 0 usando uma instrução mover comum.

```
enter_region:
        TSL REGISTER,LOCK          | copia lock para o registrador e põe lock em 1
        CMP REGISTER,#0            | lock valia zero?
        JNE enter_region           | se fosse diferente de zero, lock estaria ligado,
                                     portanto continue no laço de repetição
        RET | retorna a quem chamou; entrou na região crítica

leave_region:
        MOVE LOCK,#0               | coloque 0 em lock
        RET | retorna a quem chamou
```

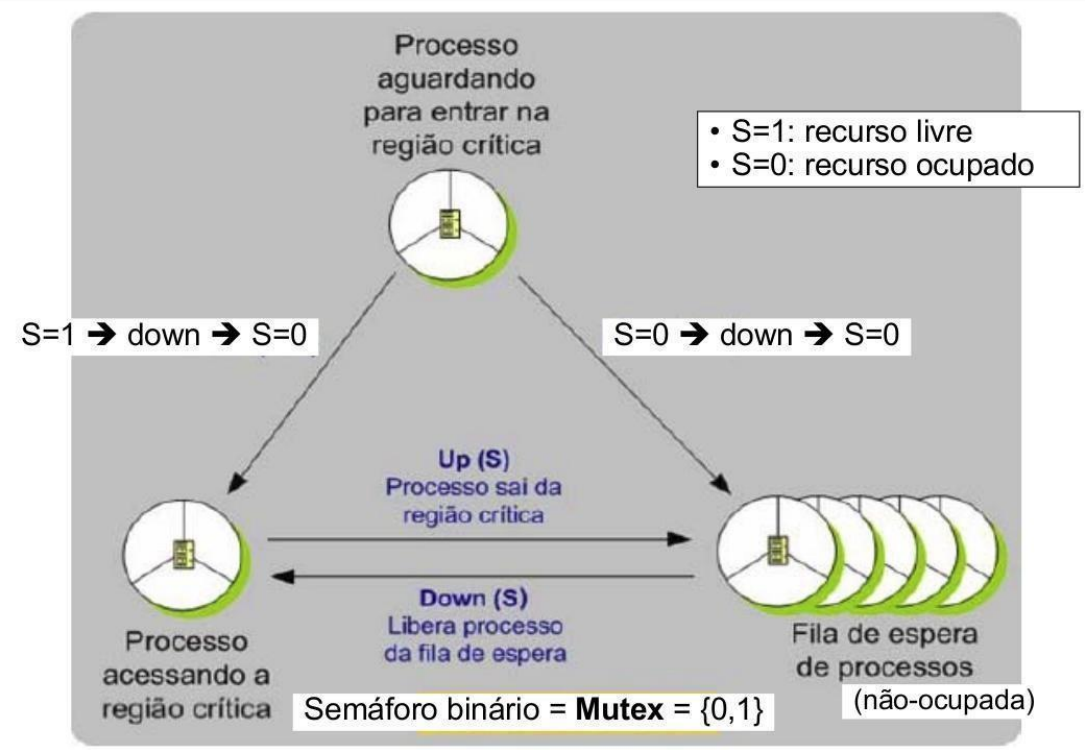
Entrando e saindo de uma região crítica usando a instrução TSL

Semáforos (S)

- Conceito proposto por E. W. Dijkstra, em 1965.
- Permite implementar, de forma simples:
  - **Exclusão mútua**
  - **Sincronização entre processos**
- Um dos principais mecanismos utilizados:
  - Nos projetos de SO e nas aplicações concorrentes.
  - Nas linguagens de programação (rotinas para semáforos).

- **Um semáforo (S) é:**
  - Uma **variável especial protegida pelo SO** (não negativa).
  - Manipulada por duas instruções atômicas:
    - ***Down*** (decremento) - generalização de *sleep*
    - ***Up*** (incremento) – generalização de *wakeup*
  - Se S=zero e *Down*: *thread* fica em espera (não ocupada).

Semáforos


Processo aguardando para entrar na região crítica
• S=1: recurso livre
• S=0: recurso ocupado
S=1 ➔ down ➔ S=0
S=0 ➔ down ➔ S=0
Up (S)
Processo sai da região crítica
Down (S)
Libera processo da fila de espera
Processo acessando a região crítica
Fila de espera de processos
(não-ocupada)
Semáforo binário = Mutex = {0,1}

Semáforos

- A operação ***down*** em um semáforo **confere para ver se o valor é maior do que 0.**
  - Se for, ele decrementará o valor (isto é, gasta um sinal de acordar armazenado) e apenas continua.
  - Se o valor for 0, o processo é colocado para dormir sem completar o *down* para o momento.
- Conferir o valor, modificá-lo e possivelmente dormir são feitos como uma única **ação atômica** indivisível.
  - Uma vez que a operação de semáforo tenha começado, nenhum outro processo pode acessar o semáforo até que a operação tenha sido concluída ou bloqueada.
  - Essencial para solucionar problemas de sincronização e evitar condições de corrida.
- A operação ***up*** incrementa o valor de um determinado semáforo.
  - Se um ou mais processos estiverem dormindo naquele semáforo, incapaz de completar uma operação *down* anterior, um deles é escolhido pelo sistema e é autorizado a completar seu *down*.
  - Após um *up* com processos dormindo em um semáforo, ele ainda estará em 0, mas haverá menos processos dormindo nele.

Mutexes

- Versão **simplificada dos semáforos**: quando a capacidade do semáforo de fazer contagem não é necessária.
- Bons somente para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhados.
- **Fáceis e eficientes de implementar** → úteis em pacotes de *threads* que são implementados inteiramente no espaço do usuário.
- Um **mutex** é uma **variável compartilhada** que pode estar em um de **dois estados**: **destravado ou travado**.
  - Apenas 1 bit é necessário para representá-lo, mas na prática muitas vezes um inteiro é usado, com 0 significando destravado e todos os outros valores significando travado.
  - **Duas rotinas** são usadas com mutexes:
    - Quando um *thread* (ou processo) precisa de acesso a uma região crítica, ele chama *mutex_lock*. Se o mutex estiver destravado naquele momento (significando que a região crítica está disponível), a chamada seguirá e o *thread* que chamou estará livre para entrar na região crítica.
    - Se o mutex já estiver travado, o *thread* que chamou será bloqueado até que o *thread* na região crítica tenha concluído e chame *mutex_unlock*.
    - Se múltiplos *threads* estiverem bloqueados no mutex, um deles será escolhido ao acaso e liberado para adquirir a trava.

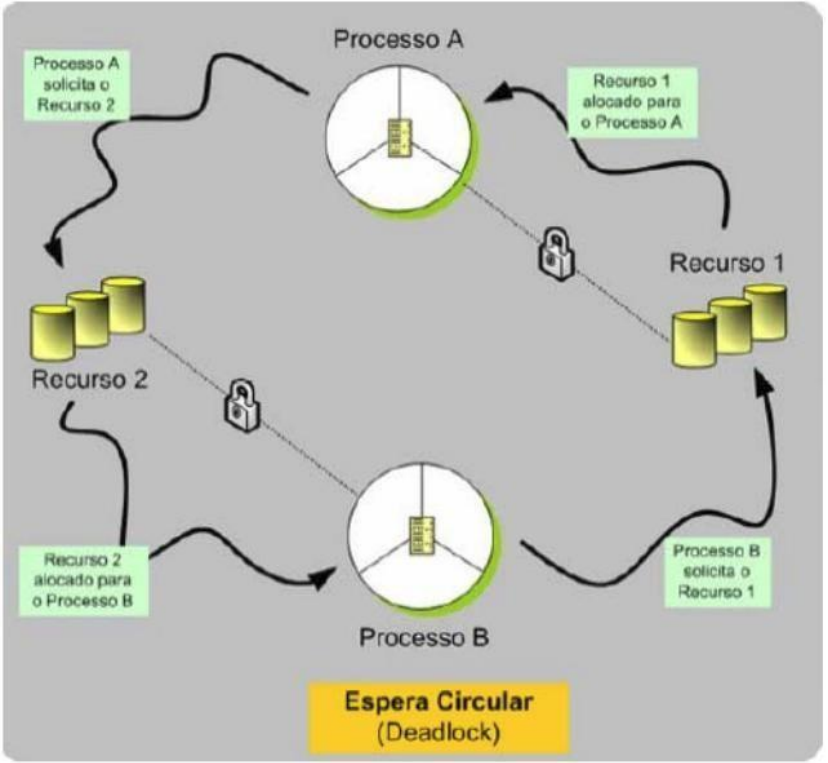
Deadlock (impasse)

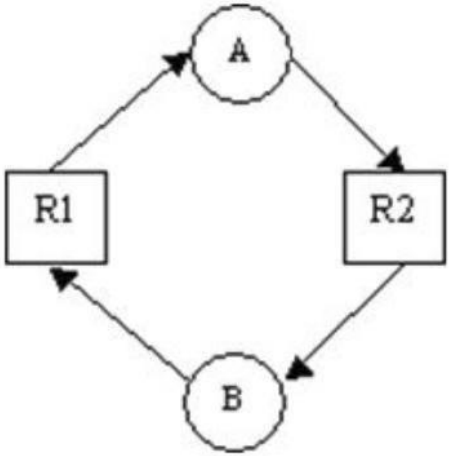
- Ocorre quando um processo aguarda por:
  - Um recurso que nunca estará disponível ou
  - Um evento que nunca ocorrerá.
- Causa provável:
  - Compartilhamento de recursos (periféricos, arquivos etc).
  - Onde a exclusão mútua é necessária.
- Coffman, Elpick e Shoshani, demonstraram, em 1971:
- Para ocorrer *deadlock*:
  - **4 condições** são essenciais (**simultâneas**).

Condições necessárias para o *Deadlock* (simultâneas)

- **Exclusão mútua:**
  - Recurso está alocado a um único processo ou
  - Está disponível.
- **Posse e espera por recurso:**
  - Processo retém um recurso (posse) e
  - Pode requisitar outro recurso (espera).
- **Não-preempção:**
  - Processo obtém um recurso (e só ele pode liberá-lo).
  - SO não tem poder de liberá-lo (quando necessário).
    - Não-preemptíveis: impressora, gravador de CD.
    - Preemptíveis: memória RAM, disco rígido.
- **Espera circular (gera um grafo cíclico de 2 ou mais processos)**
  - Processo A espera recurso 2 ora alocado ao Processo B e
  - Processo B espera recurso 1 ora alocado ao Processo A.

Deadlock (impasse)





Grafo cíclico

Soluções para o *Deadlock*

- **Prevenção:**
  - Negar **qualquer uma** das 4 condições necessárias.
- **Anulação Dinâmica:**
  - Alocar recursos criteriosamente.
- **Detecção e recuperação:** muito caro
  - Deixar ocorrer, detecta e elimina:
    - Reverter o estado do processo (*rollback*).
    - Tomar o recurso de um dos processos (preempção).
    - Encerrar um dos processos envolvidos.
- **Ignorar (Algoritmo do Avestruz)**
  - Muitos SO modernos ignoram o *deadlock* (evento raro):
    - Windows.
    - Unix like(Linux).
    - Assim: assume-se a possibilidade de **reiniciar** o sistema.

Adiamento Indefinido (*Starvation = inanição*)

- É uma situação, também conhecida como:
  - Postergação indefinida
  - Espera indefinida
  - Inanição ou
  - *Starvation*
- Processo **nunca** consegue:
  - **Executar sua região crítica**, ou seja
  - **Acessar o recurso compartilhado.**

Adiamento Indefinido (*Starvation = inanição*)

- Por que isso ocorre?
  - Quando o recurso desejado é liberado.
  - SO sempre escalona **outro** processo para o recurso.
- Solução: técnica de envelhecimento - ***aging***:
  - Elevando a **prioridade do processo** gradativamente.
  - Enquanto ele espera por um recurso.

Ler!!!

- Capítulo 2 do Livro Texto.

Atividade

- **Atividade no AVA.**