

# SISTEMAS OPERACIONAIS

## AULA 04: COMUNICAÇÃO E IMPASSES DE PROCESSOS

28 de maio de 2025

Prof. Me. José Paulo Lima

IFPE Garanhuns



**INSTITUTO  
FEDERAL**  
Pernambuco

# SUMÁRIO I



## Comunicação entre processos

- Condição de corrida

- Regiões críticas

- Exclusão mútua

  - Desabilitando interrupções

  - Variáveis do tipo trava (*lock*)

  - Chaveamento obrigatório

  - Solução de Peterson

- Semáforos

- Monitores

- Troca de mensagens

# SUMÁRIO II



## Impasses

- Definição

- Recursos

- Lidando com impasses

  - Ignorar o problema

  - Deteção e recuperação

  - Evitar dinamicamente

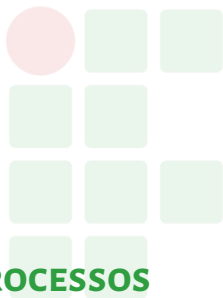
  - Algoritmo do banqueiro

- Impasses de comunicação

- Livelock*

- Inanição

## Referências



**COMUNICAÇÃO ENTRE PROCESSOS**

**INSTITUTO  
FEDERAL**  
Pernambuco

# COMUNICAÇÃO ENTRE PROCESSOS



- ▶ Frequentemente processos precisam se comunicar com outros;
- ▶ Tópicos em relação a comunicação:
  - ▶ Como um processo passar informação para outro?
  - ▶ Como garantir que processos não entrem em conflito?
  - ▶ Sequência de execução e sincronização entre processos dependentes:
    - ▶ Se o processo A produz dados e o processo B imprime. O processo B deve esperar até que A produza algum dado antes de iniciar a impressão.
- ▶ Estas questões se aplicam a threads?
  - ▶ A comunicação entre threads é similar.

# COMUNICAÇÃO ENTRE PROCESSOS



- ▶ Essas primitivas são usadas para assegurar que jamais dois processos estejam em suas regiões críticas ao mesmo tempo;
- ▶ Um processo pode estar sendo executado, ser executável, ou bloqueado, e pode mudar de estado quando ele ou outro executar uma das primitivas de comunicação entre processos.

# COMUNICAÇÃO ENTRE PROCESSOS

## CONDIÇÃO DE CORRIDA



1. A lê in o valor 7;
2. Interrupção de A  $\rightarrow$  B;
3. B lê in o valor 7;
4. Ambos processos pensam que a vaga disponível é 7;
5. Processo B executa e armazena o arquivo na posição 7 e atualiza a vaga para 8;

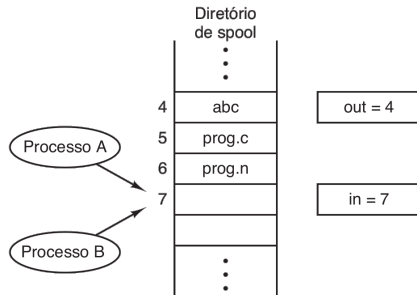


Figura extraída de Tanenbaum e Bos (2016, p. 82).

6. A executa novamente e armazena um arquivo em 7 e atualiza a próxima vaga para 8;
7. O spool está consistente mas o processo B nunca imprimirá.

# COMUNICAÇÃO ENTRE PROCESSOS

## CONDIÇÃO DE CORRIDA



- ▶ Situação onde o resultado de algum procedimento depende da ordem em que os processos executaram;
  - ▶ Altíssimos problemas de depuração/manutenção do código;
  - ▶ Programa não determinístico.
- ▶ Compartilhamento de recursos:
  - ▶ Memória principal
    - ▶ Compartilhamento de variáveis.
  - ▶ Memória secundária
    - ▶ Arquivo compartilhado.



# COMUNICAÇÃO ENTRE PROCESSOS

## REGIÕES CRÍTICAS



### Definição:

“A parte do programa onde é feito o acesso ao recurso compartilhado é denominada região crítica (*critical region*). Se for possível evitar que dois processos entrem em suas regiões críticas ao mesmo tempo, ou seja, se for garantida a execução mutuamente exclusiva das regiões críticas, os problemas decorrentes do compartilhamento serão evitados.” (MACHADO; MAIA, 2017, p. 100).

# COMUNICAÇÃO ENTRE PROCESSOS

## REGIÕES CRÍTICAS

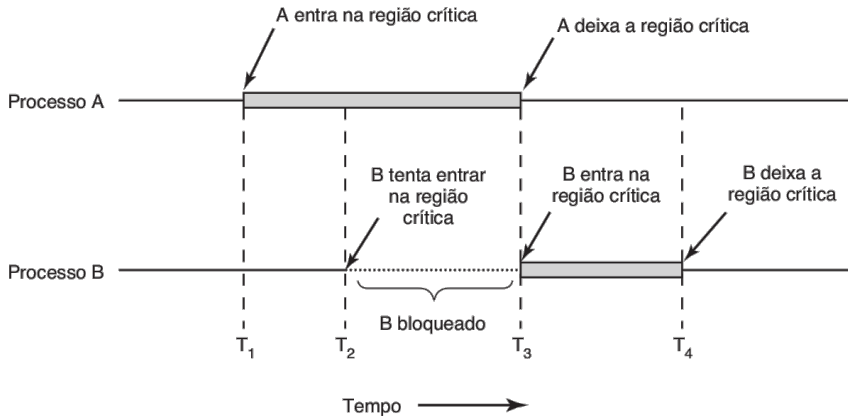


Figura extraída de Tanenbaum e Bos (2016, p. 83).

# COMUNICAÇÃO ENTRE PROCESSOS

## EXCLUSÃO MÚTUA



- ▶ O que fazer para evitar condições de disputa?
  - ▶ A solução é encontrar algum modo de impedir que mais de um processo leia e escreva ao mesmo tempo na memória compartilhada;
    - ▶ Exclusão mútua: Impedir que um processo use um recurso compartilhado em uso por outro processo.
  - ▶ Precisamos satisfazer quatro condições:
    1. Dois processos nunca podem estar simultaneamente em suas regiões críticas;
    2. Nada pode ser afirmado sobre a velocidade ou sobre o número de CPU's;
    3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos;
    4. Nenhum processo deve esperar eternamente para entrar em sua região crítica.

# COMUNICAÇÃO ENTRE PROCESSOS

## EXCLUSÃO MÚTUA



### Definição

“A solução mais simples para evitar os problemas de compartilhamento [...] é impedir que dois ou mais processos acessem um mesmo recurso simultaneamente. Para isso, enquanto um processo estiver acessando determinado recurso, todos os demais processos que queiram acessá-lo deverão esperar pelo término da utilização do recurso. Essa ideia de exclusividade de acesso é chamada exclusão mútua (*mutual exclusion*).” (MACHADO; MAIA, 2017, p. 100).

► Exclusão mútua com espera ociosa:

1. Desabilitando interrupções;
2. Variáveis do tipo trava (*lock*);
3. Chaveamento obrigatório;
4. Solução de Peterson.

# COMUNICAÇÃO ENTRE PROCESSOS

## EXCLUSÃO MÚTUA: DESABILITANDO INTERRUPÇÕES



- ▶ Nenhum processo poderá ser interrompido antes de sair da região crítica;
- ▶ Não é uma boa abordagem;
  - ▶ Processos usuários não poderiam ter esse privilégio de desligar interrupções;
    - ▶ Se um processo usuário a tenha desligado e nunca mais ligado, todo o sistema estaria prejudicado.
  - ▶ Em sistemas multiprocessadores, desabilitar interrupções afetará somente aquela CPU que executou a interrupções “Disable”.
- ▶ Vantagem:
  - ▶ O núcleo pode desabilitar interrupções para algumas funções.
    - ▶ Atualizando a tabela de processos prontos.
- ▶ Desabilitar interrupções muitas vezes é útil dentro do sistema operacional mas inadequada para processos usuários;
- ▶ Em sistemas multicore, desabilitar as interrupções não impede que outras CPU's interfiram nas operações que a primeira CPU está executando.

# COMUNICAÇÃO ENTRE PROCESSOS

## EXCLUSÃO MÚTUA: VARIÁVEIS DO TIPO TRAVA (LOCK)



- ▶ Solução de software;
- ▶ É a utilização de uma variável *Lock* para controlar quem entra na região crítica;
- ▶ Variável compartilhada funciona como trava;
  - 0 Está livre;
  - 1 Está bloqueada;
  - ▶ Funcionamento:
    - ▶ Inicialmente *Lock* contém o valor 0;
    - ▶ Para o processo entrar na região crítica, verifica se o valor de *Lock*;
    - ▶ Se for 0, o processo altera essa variável para 1 e entra na região crítica;
    - ▶ Se for 1, o processo simplesmente fica aguardando até ela se tornar 0.
- ▶ Problema idêntico ao do diretório de *spool*.
  - ▶ Antes de um processo possa alterar a variável *lock* para 1, outro processo é escalonado, executa e altera a variável *lock* para 1. O primeiro processo colocará 1 também na variável *lock*, assim, os dois processos estão em suas regiões críticas ao mesmo tempo.

# COMUNICAÇÃO ENTRE PROCESSOS

## EXCLUSÃO MÚTUA: CHAVEAMENTO OBRIGATÓRIO

- ▶ Chavear a vez não é uma boa ideia quando um dos processos for muito mais lento que o outro;
- ▶ Turn inicialmente é 0;
- ▶ Processo 0 entra na região crítica;
- ▶ Processo 0 sai, configura turn=1 e permite Processo 1 entrar na região crítica;
- ▶ Se o Processo 1 termina rápido, ambos os processos estão em suas regiões não críticas.

### Processo 0:

```
1 while (TRUE){  
2     while (turn !=0)  
3         critical_region(); // laço  
4     turn = 1;  
5     noncritical_region();  
6 }
```

### Processo 1:

```
1 while (TRUE){  
2     while (turn !=1)  
3         critical_region(); // laço  
4     turn = 0;  
5     noncritical_region();  
6 }
```

# COMUNICAÇÃO ENTRE PROCESSOS

## EXCLUSÃO MÚTUA: SOLUÇÃO DE PETERSON



### Definição:

“Ao combinar a ideia de alternar a vez com a ideia das variáveis de trava e de advertência, um matemático holandês, T. Dekker, foi o primeiro a desenvolver uma solução de software para o problema da exclusão mútua que não exige uma alternância explícita.” (TANENBAUM; BOS, 2016, p. 85).

“Antes de acessar a região crítica, o processo sinaliza esse desejo através da variável de condição, porém o processo cede o uso do recurso ao outro processo [...]. Em seguida o comando WHILE como protocolo de entrada da região crítica.” (MACHADO; MAIA, 2017, p. 109).



# COMUNICAÇÃO ENTRE PROCESSOS

## SEMÁFOROS



### Definição:

“O conceito de semáforos foi proposto por E. W. Dijkstra em 1965, sendo apresentado como um mecanismo que permitia implementar, de forma simples a exclusão mútua e sincronização condicional entre processos. [...]

Os semáforos podem ser classificados como binários ou contadores. Os semáforos binários, também chamados de mutexes (*mutual exclusion semaphores*), só podem assumir os valores 0 e 1, enquanto os semáforos contadores podem assumir qualquer valor inteiro positivo, além do 0.” (MACHADO; MAIA, 2017, p. 112).

# COMUNICAÇÃO ENTRE PROCESSOS

## SEMÁFOROS

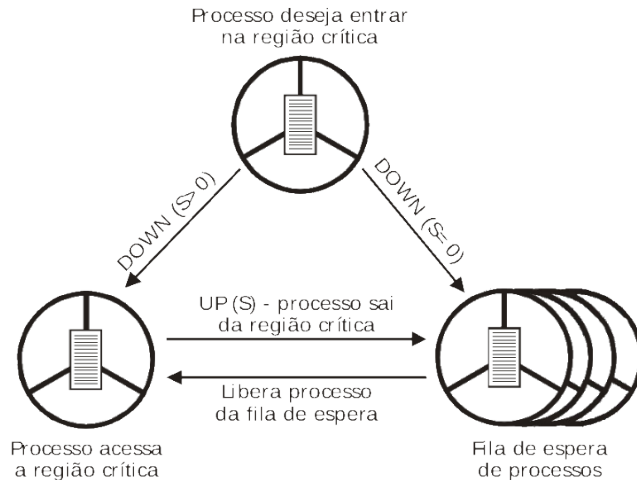


Figura extraída de Machado e Maia (2017, p. 113).

# COMUNICAÇÃO ENTRE PROCESSOS

## SEMÁFOROS



- ▶ Exclusão mútua utilizando semáforos;
  - ▶ Geralmente utilizado um Mutex.
- ▶ Sincronização condicional utilizando semáforos;
- ▶ Variável capaz de armazenar um valor maior ou igual a zero:
  - ▶ down
    - ▶ Verifica o valor do semáforo;
    - ▶ Se for zero, dorme;
    - ▶ Senão decrementa o valor e prossegue;
    - ▶ Operação atômica e indivisível.
  - ▶ up
    - ▶ Se tiver processo dormindo, acorda um deles;
    - ▶ Senão, incrementa a variável;
    - ▶ Também é atômica e indivisível.
- ▶ Existem outras versões clássicas como:
  - ▶ Problema do Jantar dos Filósofos;
  - ▶ Problema do Barbeiro.

# COMUNICAÇÃO ENTRE PROCESSOS

## MONITORES



### Definição:

“Um monitor é uma coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo ou pacote. Processos podem chamar as rotinas em um monitor sempre que eles quiserem, mas eles não podem acessar diretamente as estruturas de dados internos do monitor a partir de rotinas declaradas fora dele.” (TANENBAUM; BOS, 2016, p. 96).

# COMUNICAÇÃO ENTRE PROCESSOS

## MONITORES

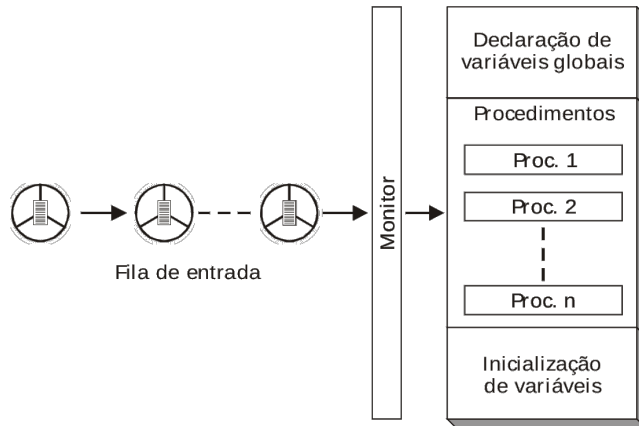


Figura extraída de Machado e Maia (2017, p. 120).

# COMUNICAÇÃO ENTRE PROCESSOS

## MONITORES



- ▶ Os monitores são construções de linguagens de programação que fornecem uma funcionalidade equivalente aos semáforos;
  - ▶ Mais fácil de controlar.
- ▶ O monitor é um conjunto de procedimentos, variáveis e inicialização definidos dentro de um módulo;
- ▶ A característica mais importante do monitor é a exclusão mútua automática entre os seus procedimentos;
  - ▶ Basta codificar as regiões críticas como procedimentos do monitor e o compilador irá garantir a exclusão mútua;
  - ▶ Desenvolvimento é mais fácil;
  - ▶ Existem linguagens que não possuem monitores;
  - ▶ Os monitores são um conceito de linguagem de programação.

# COMUNICAÇÃO ENTRE PROCESSOS

## MONITORES



- ▶ Para implementar a sincronização é necessário utilizar variáveis de condição;
  - ▶ São tipos de dados especiais dos monitores;
  - ▶ São operadas por duas instruções `Wait` e `Signal`:
    - `Wait(C)` Suspende a execução do processo, colocando-o em estado de espera associado a condição `C`;
    - `Signal(C)` Permite que um processo bloqueado por `Wait(C)` continue a sua execução.
      - ▶ Se existir mais de um processo bloqueado, apenas um é liberado;
      - ▶ Se não existir nenhum processo bloqueado, não faz nada.

# COMUNICAÇÃO ENTRE PROCESSOS

## TROCA DE MENSAGENS



### Definição:

“Esse método de comunicação entre processos usa duas primitivas, send e receive, que, como semáforos e diferentemente dos monitores, são chamadas de sistema em vez de construções de linguagem. Como tais, elas podem ser facilmente colocadas em rotinas de biblioteca”. (TANENBAUM; BOS, 2016, p. 99).



# COMUNICAÇÃO ENTRE PROCESSOS

## TROCA DE MENSAGENS



- ▶ Muito mais complexo do que as outras abordagens;
- ▶ Diversos problemas podem ocorrer.
  - ▶ Perda de mensagem
    - ▶ Máquinas conectadas por uma rede;
    - ▶ Solução: ACK;
      - Caso o ACK se perder, o emissor emite mais de uma vez a mesma mensagem;
      - Receptor deve saber distinguir mensagens novas de mensagens repetidas;
      - Número de sequência.
  - ▶ Problemas de autenticação;
  - ▶ Custo para troca de mensagens é muito alto.
- ▶ Troca de mensagens através do `send(destination, &message)` e `receive(source, &message)`.
  - ▶ Chamadas do sistema;
  - ▶ Não são vinculados a linguagens de programa.

# COMUNICAÇÃO ENTRE PROCESSOS

## TROCA DE MENSAGENS: COMUNICAÇÃO DIRETA

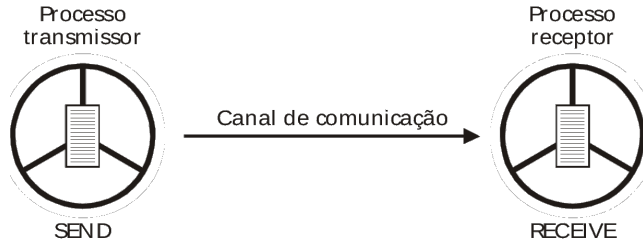


Figura extraída de Machado e Maia (2017, p. 125).

# COMUNICAÇÃO ENTRE PROCESSOS

## TROCA DE MENSAGENS: COMUNICAÇÃO INDIRETA

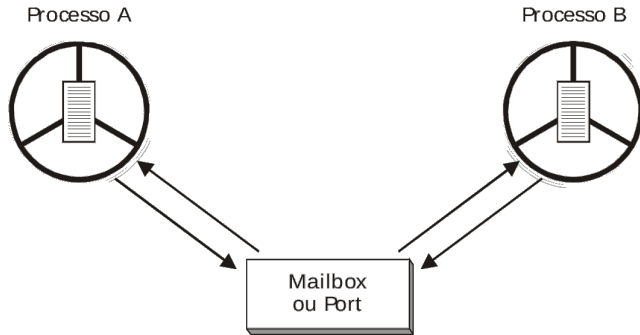
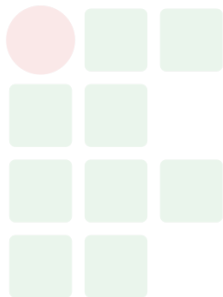


Figura extraída de Machado e Maia (2017, p. 126).

**IMPASSES**



**INSTITUTO  
FEDERAL**  
Pernambuco

# IMPASSES

## DEFINIÇÃO



### Definição:

Impasse ou “*Deadlock*” é a situação em que um processo aguarda por um recurso que nunca estará disponível ou um evento que não ocorrerá.” (MACHADO; MAIA, 2017, p. 127).

“Um conjunto de processos estará em situação de impasse se cada processo no conjunto estiver esperando por um evento que apenas outro processo no conjunto pode causar.” (TANENBAUM; BOS, 2016, p. 303).

# IMPASSES

## DEFINIÇÃO

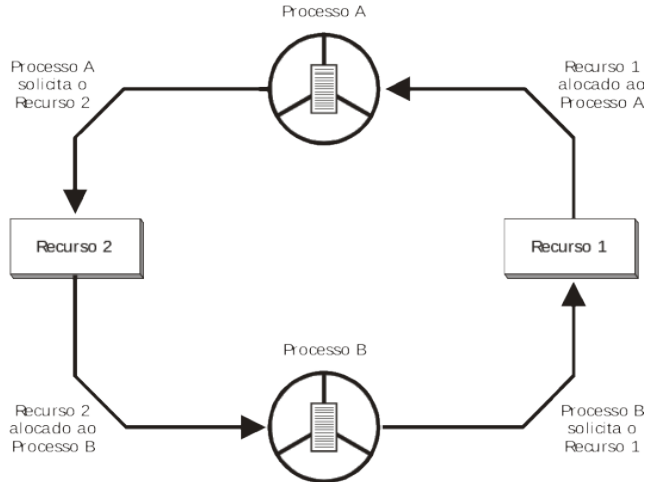


Figura extraída de Machado e Maia (2017, p. 128).

# IMPASSES

## RECURSOS



- ▶ Recursos preemptíveis
  - ▶ “Um recurso preemptível é aquele que pode ser retirado do processo proprietário sem causar-lhe prejuízo algum.” (TANENBAUM; BOS, 2016, p. 302);
  - ▶ Exemplo: memória.
- ▶ Recursos não-preemptíveis
  - ▶ “[...] é um recurso que não pode ser tomado do seu proprietário atual sem potencialmente causar uma falha.” (TANENBAUM; BOS, 2016, p. 302);
  - ▶ Exemplo: Gravador de mídias (CD, DVD, Blu-ray).

### Destaque:

- ▶ Em geral, impasses envolvem recursos não preemptíveis;
- ▶ Impasses que envolvem recursos preemptíveis são solucionados realocando recursos entre eles.

# IMPASSES

## EXEMPLO PRÁTICO



- ▶ Tome, como exemplo, dois processos que queiram cada um gravar um documento escaneado em um disco Blu-ray;
  1. O processo A solicita permissão para usar o scanner e ela lhe é concedida;
  2. O processo B é programado diferentemente e solicita o gravador Blu-ray primeiro e ele também lhe é concedido;
  3. Agora A pede pelo gravador Blu-ray, mas a solicitação é suspensa até que B o libere;
  4. Infelizmente, em vez de liberar o gravador Blu-ray, B pede pelo scanner.
- ▶ Ambos os processos estão bloqueados e assim permanecerão para sempre.
  - ▶ Essa situação é chamada de impasse (*deadlock*).



# IMPASSES

## CONDIÇÕES PARA OCORRÊNCIA DE IMPASSES



Coffman, Elphick e Shoshani (1971) apontam quatro condições válidas para haver um impasse de recurso:

1. Condição de exclusão mútua:
  - ▶ Cada recurso está atualmente associado a exatamente um processo ou está disponível.
2. Condição de posse e espera:
  - ▶ Processos atualmente de posse de recursos que foram concedidos antes podem solicitar novos recursos.
3. Condição de não preempção:
  - ▶ Recursos concedidos antes não podem ser tomados à força de um processo. Eles precisam ser explicitamente liberados pelo processo que os têm.
4. Condição de espera circular:
  - ▶ Deve haver uma lista circular de dois ou mais processos, cada um deles esperando por um processo de posse do membro seguinte da cadeia.

# IMPASSES

## LIDANDO COM IMPASSES



Em geral, quatro estratégias são usadas para lidar com impasses:

1. Ignorar o problema;
2. Detecção e recuperação;
3. Evitar dinamicamente pela alocação cuidadosa de recursos;
4. Prevenção, ao negar estruturalmente uma das quatro condições.

# LIDANDO COM IMPASSES

## IGNORAR O PROBLEMA



- ▶ Comparar a frequência de ocorrência de *deadlocks* com a frequência de outras falhas do sistema;
  - ▶ Falhas de hardware, erros de compiladores, erros do Sistema Operacional, etc.
- ▶ Se o esforço em solucionar o problema for muito grande em relação a frequência do *deadlock*, então ele pode ser ignorado;
- ▶ Algoritmo do Avestruz.
  - ▶ Finge que o problema não existe.

# LIDANDO COM IMPASSES

## DETECÇÃO E RECUPERAÇÃO



- ▶ Recuperação através de preempção:
  - ▶ Possibilidade de retirar temporariamente um recurso de seu atual dono (processo) e entregá-lo a outro processo.
- ▶ Recuperação através de *rollback*:
  - ▶ O estado de cada processo (e recurso por ele usado) é periodicamente armazenado em um arquivo de verificação (*checkpoint file*);
  - ▶ Quando ocorre um *deadlock*, o processo que detém o recurso é voltado a um ponto antes de adquirir esse recurso (via o *checkpoint file* apropriado).
- ▶ Recuperação através de eliminação (*kill*):
  - ▶ Um ou mais processos que estão no ciclo com *deadlock* são interrompidos.

# LIDANDO COM IMPASSES

## DETECÇÃO E RECUPERAÇÃO

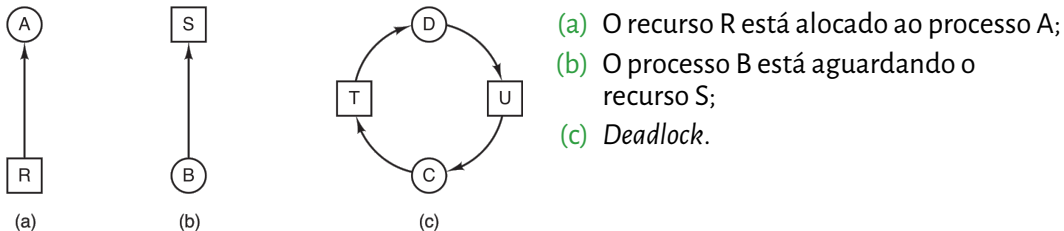


Figura extraída de Tanenbaum e Bos (2016, p. 304).

# LIDANDO COM IMPASSES

## DETECÇÃO E RECUPERAÇÃO

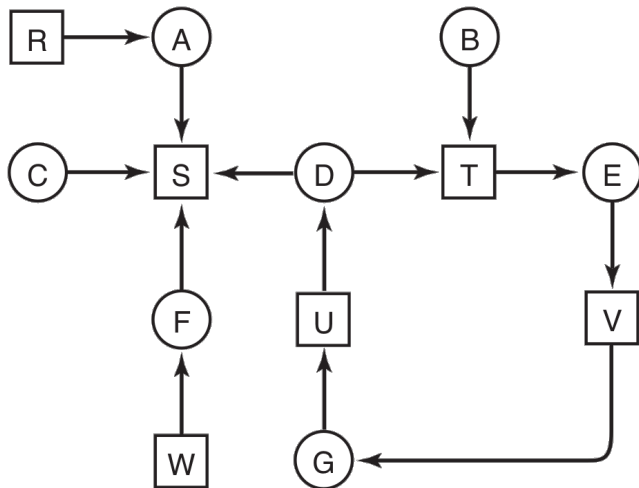


Figura extraída de Tanenbaum e Bos (2016, p. 304).

# LIDANDO COM IMPASSES

## EVITAR DINAMICAMENTE



- ▶ Alocação de recursos conforme a necessidade;
- ▶ Escalonamento cuidadoso (alto custo);
- ▶ Conhecimento dos recursos que serão utilizados;
- ▶ Algoritmos:
  - ▶ Banqueiro para um único tipo de recurso;
  - ▶ Banqueiro para vários tipos de recurso.
- ▶ Usam a noção de Estados Seguros e Inseguros.

# LIDANDO COM IMPASSES

## ESTADOS SEGUROS E INSEGUROS



- ▶ Usa as mesmas estruturas de detecção com vários recursos;
- ▶ Um estado consiste das estruturas:

**E** *Existing*;  
**A** *Available*;

**C** *Current*;  
**R** *Requirement*.

- ▶ Estados Seguros
  - ▶ Não provocam *deadlocks*;
  - ▶ Há uma maneira de atender a todas as requisições;
    - ▶ Ainda que todos peçam seu número máximo de recursos imediatamente.
  - ▶ A partir de um estado seguro, existe a garantia de que todos os processos terminarão.
- ▶ Estados Inseguros
  - ▶ Podem provocar *deadlocks*;
  - ▶ Mas não necessariamente provocam;
  - ▶ A partir de um estado inseguro, não é possível garantir que os processos terminarão corretamente.



# LIDANDO COM IMPASSES

## ESTADOS SEGUROS E INSEGUROS - EXEMPLO



- ▶ A tem 3 instâncias, mas pode precisar de até 9;
- ▶ B tem 2, e pode precisar de 4;
- ▶ C tem 2, e pode precisar de 7;
- ▶ Havendo um total de 10 instâncias do recurso, diante dessas necessidades, sobrarão 3.

Processo	Alocado	Máximo
A	3	9
B	2	4
C	2	7

↑ Livre = 3

# LIDANDO COM IMPASSES

## ESTADOS SEGUROS E INSEGUROS - EXEMPLO



► Sequência para que todos os processos terminem:

1. O escalonador roda B até que ele peça mais recursos;

Processo	Alocado	Máximo
A	3	9
B	<del>2</del> → 4	4
C	2	7

Livre = 1

# LIDANDO COM IMPASSES

## ESTADOS SEGUROS E INSEGUROS - EXEMPLO



- Sequência para que todos os processos terminem:

2. B termina a execução e libera os recursos;

Processo	Alocado	Máximo
A	3	9
B	0	-
C	2	7

Livre = 5

# LIDANDO COM IMPASSES

## ESTADOS SEGUROS E INSEGUROS - EXEMPLO



- ▶ Sequência para que todos os processos terminem:
  - 3. C aloca recursos para sua execução e roda até terminar;

Processo	Alocado	Máximo
A	3	9
B	0	-
C	<del>2</del> → 7	7

Livre = 0

# LIDANDO COM IMPASSES

## ESTADOS SEGUROS E INSEGUROS - EXEMPLO



- Sequência para que todos os processos terminem:

4. Agora o processo A pode alocar as 6 instâncias que necessitava para terminar;

Processo	Alocado	Máximo
A	<del>3</del> → 9	9
B	0	-
C	0	-

Livre = 1

# LIDANDO COM IMPASSES

## ESTADOS SEGUROS E INSEGUROS



- ▶ Idealizado por Dijkstra (1968);
- ▶ Algoritmo de escalonamento para evitar *deadlock*;
- ▶ Funciona como um banqueiro (SO):
  - ▶ Dar crédito (recursos) aos clientes (processos).
- ▶ Verifica se um pedido leva a um estado inseguro.
- ▶ Se levar, o pedido é negado, se não, é executado.

# LIDANDO COM IMPASSES

## ALGORITMO DO BANQUEIRO

- ▶ Algoritmo do banqueiro para vários recursos;
- ▶ Mesma ideia, mas duas matrizes são utilizadas;
- ▶ Exemplos:

- ▶ Recursos:

**F** Unidade de Fita - 6;

**P** Plotters - 3;

**I** Impressoras - 4;

**C** Unidade de CD-ROM - 2.

- ▶ Quantidade alocada:

- ▶ (5, 3, 2, 2)

- ▶ Recursos disponíveis:

- ▶ (1, 0, 2, 0)

### Recursos alocado

Processos	F	P	I	C
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

### Necessita alocar

Processos	F	P	I	C
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

# LIDANDO COM IMPASSES

## ALGORITMO DO BANQUEIRO

1. É fácil observar que podemos alocar recurso para executar primeiramente o processo D;

- ▶ Quantidade alocada:
  - ▶ (5, 3, 3, 2)

### Recursos alocado

Processos	F	P	I	C
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	1	1
E	0	0	0	0

- ▶ Recursos disponíveis:
  - ▶ (1, 0, 1, 0)

### Necessita alocar

Processos	F	P	I	C
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	0	0
E	2	1	1	0



# LIDANDO COM IMPASSES

## ALGORITMO DO BANQUEIRO

### 2. O processo D finaliza e libera os recursos;

#### ► Quantidade alocada:

► (4, 2, 2, 1)

#### Recursos alocado

Processos	F	P	I	C
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	0	0	0	0
E	0	0	0	0

#### ► Recursos disponíveis:

► (2, 1, 2, 1)

#### Necessita alocar

Processos	F	P	I	C
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	-	-	-	-
E	2	1	1	0

# LIDANDO COM IMPASSES

## ALGORITMO DO BANQUEIRO



3. Após a execução de D, podemos executar tanto os processos A ou E. Vamos primeiramente executar ou E?

► Quantidade alocada:

► (6, 3, 3, 1)

Recursos alocado

Processos	F	P	I	C
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	0	0	0	0
E	2	1	1	0

► Recursos disponíveis:

► (0, 0, 1, 1)

Necessita alocar

Processos	F	P	I	C
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	-	-	-	-
E	0	0	0	0

# LIDANDO COM IMPASSES

## ALGORITMO DO BANQUEIRO



### 4. Após a execução de E, há a liberação de recursos;

#### ► Quantidade alocada:

► (4, 2, 2, 1)

#### Recursos alocado

Processos	F	P	I	C
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	0	0	0	0
E	0	0	0	0

#### ► Recursos disponíveis:

► (2, 1, 2, 1)

#### Necessita alocar

Processos	F	P	I	C
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	-	-	-	-
E	-	-	-	-

# LIDANDO COM IMPASSES

## ALGORITMO DO BANQUEIRO

5. Colocando A para executar, e após a execução temos a liberação dos recursos;

► Quantidade alocada:

► (1, 2, 1, 1)

Recursos alocado

Processos	F	P	I	C
A	0	0	0	0
B	0	1	0	0
C	1	1	1	0
D	0	0	0	0
E	0	0	0	0

► Recursos disponíveis:

► (6, 2, 1, 2)

Necessita alocar

Processos	F	P	I	C
A	0	0	0	0
B	0	1	1	2
C	3	1	0	0
D	-	-	-	-
E	-	-	-	-

6. Após isso teremos recursos para executar B e C.

# IMPASSES DE COMUNICAÇÃO



- ▶ Outro tipo de impasse pode ocorrer em sistemas de comunicação em que dois ou mais processos comunicam-se enviando mensagens;
- ▶ Um arranjo comum é:
  - ▶ O processo A enviar uma mensagem de solicitação ao processo B, e então bloquear até B enviar de volta uma mensagem de resposta;
  - ▶ Suponha que a mensagem de solicitação se perca;
  - ▶ A está bloqueado esperando pela resposta;
  - ▶ B está bloqueado esperando por uma solicitação pedindo a ele para fazer algo.

## Definição:

“Em algumas situações, um processo abre mão dos bloqueios que ele já adquiriu quando nota que não pode obter o bloqueio seguinte de que precisa. Ele espera e tenta de novo. [...] Mas, se o outro processo faz a mesma coisa exatamente no mesmo momento, eles estarão na situação de duas pessoas tentando passar uma pela outra quando ambas educadamente dão um passo para o lado e, no entanto, nenhum progresso é possível.” (TANENBAUM; BOS, 2016, p. 318).

- ▶ É claro, nenhum processo é bloqueado e poderíamos até dizer que as coisas estão acontecendo, então isso não é um impasse;
- ▶ Ainda assim, nenhum progresso é possível, então temos algo equivalente: um *livelock*.

# INANIÇÃO

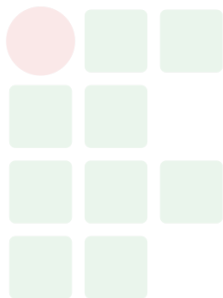


## Definição:

“Solicitações para recursos acontecem o tempo todo. Alguma política é necessária para tomar uma decisão sobre quem recebe qual recurso e quando. Essa política, embora aparentemente razoável, pode levar a alguns processos nunca serem servidos, embora não estejam em situação de impasse.” (TANENBAUM; BOS, 2016, p. 319).

- ▶ Inanição é morrer de fome;
- ▶ Um problema relacionado com o impasse e o *livelock* é a inanição (*starvation*);
- ▶ A inanição pode ser evitada com uma política de alocação de recursos primeiro a chegar, primeiro a ser servido.





## REFERÊNCIAS

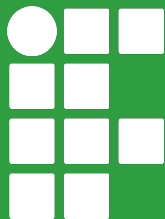


INSTITUTO  
FEDERAL  
Pernambuco



## REFERÊNCIAS I

-  COFFMAN, E. G.; ELPHICK, M.; SHOSHANI, A. System deadlocks. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 3, n. 2, p. 67–78, jun. 1971. ISSN 0360-0300. Disponível em: <<http://doi.acm.org/10.1145/356586.356588>>. Acesso em: 24 abr. 2018.
-  DIJKSTRA, Edsger W. Cooperating sequential processes. In: \_\_\_\_\_. **The Origin of Concurrent Programming**: From semaphores to remote procedure calls. New York: Springer, 1968. p. 65–138. ISBN 978-1-4419-2986-0.
-  MACHADO, Francis Berenger; MAIA, Luiz Paulo. **Arquitetura de Sistemas Operacionais**. 5. ed. Rio de Janeiro: LTC, 2017. ISBN 978-85-216-2210-9.
-  TANENBAUM, Andrew S.; BOS, Herbert. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Pearson Education do Brasil, 2016.



**INSTITUTO FEDERAL**

Pernambuco