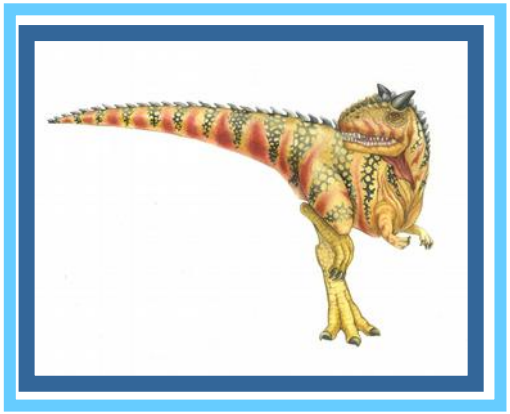
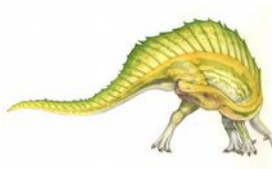
Capítulo 7: Deadlocks



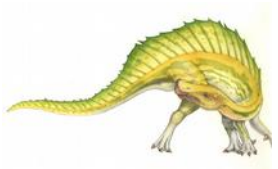
Sumário

- Modelo de sistema considerado
- *Deadlocks*
- Métodos para lidar com *deadlocks*:
  - prevenção
  - impedimento
  - detecção
  - recuperação

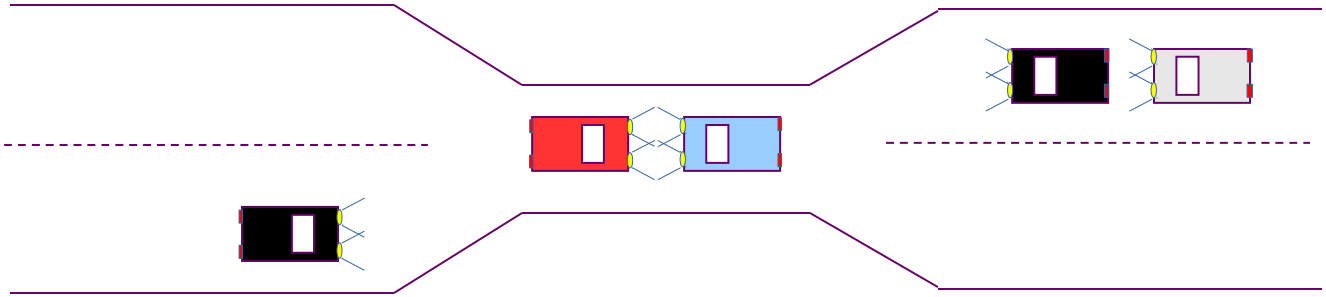


O problema de *deadlock*

- Um conjunto de processos bloqueados, cada um de posse de um recurso e esperando por outro, já obtido por algum outro processo no conjunto
  - Analogia: *"quando dois carros se aproximarem um do outro em um cruzamento, ambos devem parar completamente e nenhum dos dois deve se movimentar até que o outro passe"*
- Exemplos:
  - Um sistema com dois drives de DVD
    - P1 e P2 já reservaram um drive cada e precisam do outro
  - Semáforos A e B, inicializados em 1
    - P0 executa wait(A) e é reescalonado;
    - P1 executa wait(B); wait(A);
    - P0 reassume e executa wait(B).



Travessia de uma ponte estreita



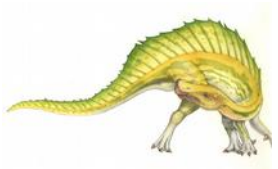
- Cada parte da ponte é vista como um recurso
- Se um bloqueio ocorre (deadlock) pode ser resolvido com um carro dando ré
  - Carro libera recursos e retrocede
  - Vários carros podem ter que fazê-lo
- Inanição é possível

Modelo do sistema

- Recursos têm vários tipos R_1, R_2, … , R_m
  - ex.: ciclos de CPU, espaço de memória, dispositivo de E/S
- Cada recurso R_i tem W_i instâncias
  - ex.: R_1=impressora, W_1={imprA, imprB, imprC}
- Processos acessam recursos da mesma forma:
  - Solicitação* → Uso → Liberação
    - * se não puder ser atendida imediatamente, espera
- Solicitação e Liberação são chamadas de sistema, ex.:
  - arquivo: open() e close()
  - memória: allocate() e free()

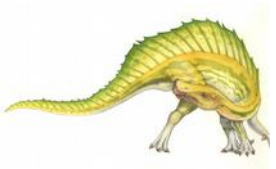
Deadlock: condições necessárias

- Exclusão mútua
  - apenas um processo por vez pode acessar o recurso
  - se outro solicitar, deve esperar até que seja liberado
- Posse durante a espera
  - processo com posse de pelo menor um recurso e esperando para adquirir recursos adicionais detidos por outros processos
- Inexistência de preempção
  - recurso só pode ser liberado voluntariamente pelo processo
- Espera circular
  - $P_0 – P_1 – P_2 – \ldots – P_n – P_0$



Grafo de alocação de recursos

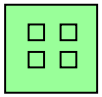
- Permite descrever com mais precisão *deadlocks*
- Vértices são divididos em dois tipos:
  - $P = \{P_1, P_2, \ldots, P_n\}$, os processos no sistema
  - $R = \{R_1, R_2, \ldots, R_m\}$, os recursos do sistema
- Arestas são também são de dois tipos:
  - solicitação: aresta direcionada Pi → Rj
  - atribuição: aresta direcionada Ri → Pj



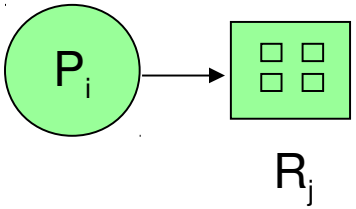
Grafo de alocação de recursos

- Um processo:

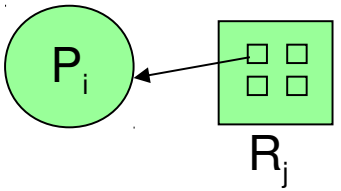
- Tipo de recurso com 4 instâncias

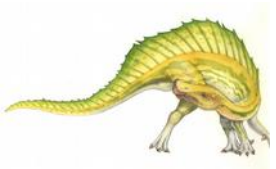


- Pi requisita instância de Rj



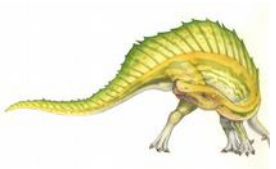
- Pi detém uma instância de Rj



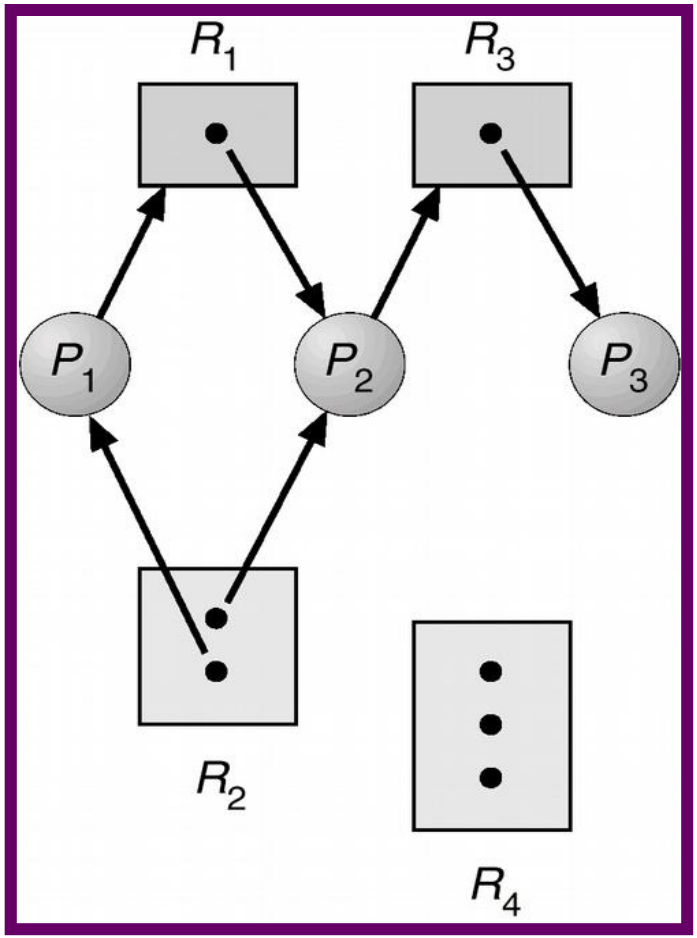


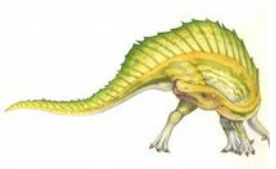
Fatos básicos

- Se não há ciclos no grafo → não há *deadlock* *:)*
- Se o grafo contém ciclos, depende:
  - se recursos só têm uma instância → há *deadlock* *: (*
    - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$
  - se há mais de uma instância → possível *deadlock* *O_o*

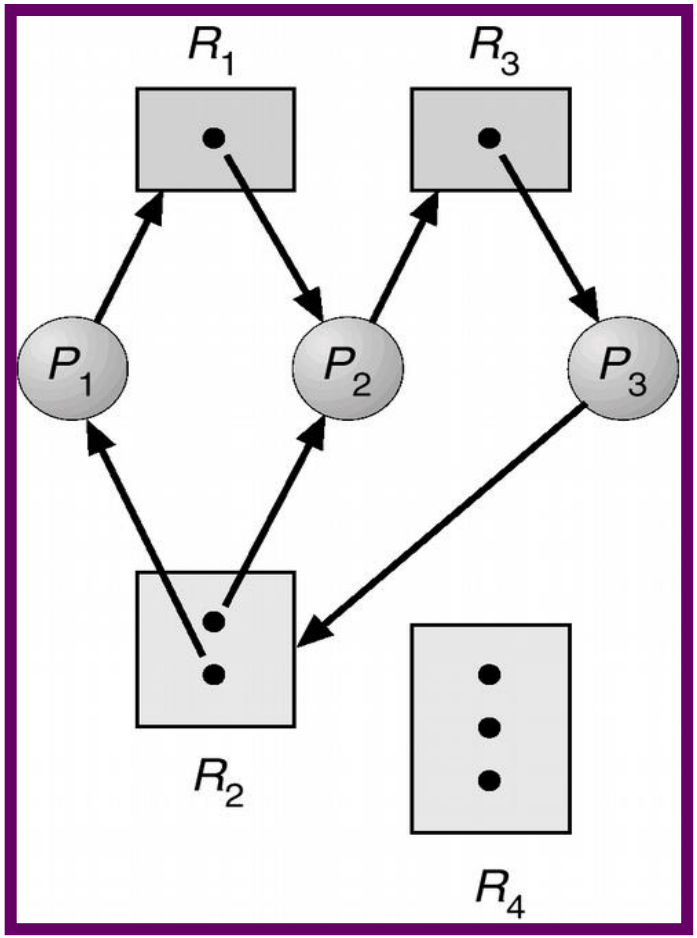


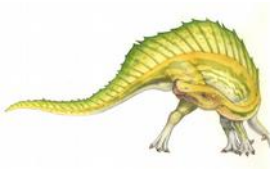
Grafo de alocação de recursos



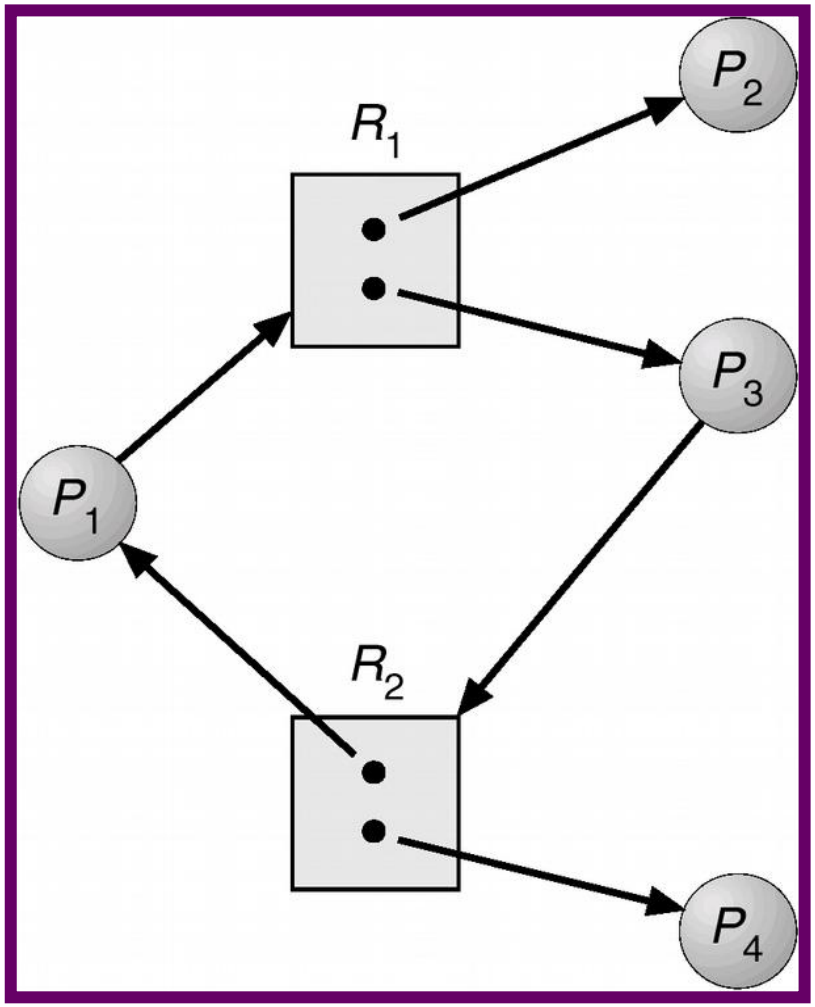


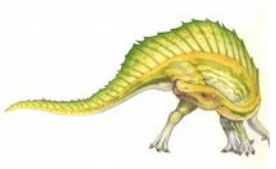
Grafo de alocação de recursos com *deadlock*





Grafo de alocação de recursos com ciclo, mas sem *deadlock*





Formas de lidar com *deadlocks*

- Use um protocolo para prevenir ou impedir a ocorrência de *deadlocks*
  - garante que o sistema NUNCA entrará em estado de *deadlock*
- Permita que o sistema entre em estado de *deadlock*, detecte-o e execute uma recuperação
- Ignore o problema e faça de conta que *deadlocks* nunca ocorrerão no sistema
  - usado na maioria dos sistemas, inclusive Unix e Windows!
  - fica a cargo do programador manipular os *deadlocks*

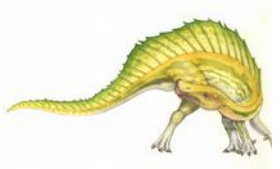
Prevenção de *deadlocks*

- Garanta que pelo menos uma das condições originais para *deadlocks* nunca ocorra
  - 1) Exclusão mútua
  - 2) Posse durante a espera
  - 3) Inexistência de preempção
  - 4) Espera circular
- Se ao menos uma dessas condições não possa acontecer, previne contra a ocorrência de *deadlocks*
- Para tal, restringe as formas como as solicitações de recursos podem ser feitas

Prevenção de *deadlocks*

- Garanta que pelo menos uma das condições originais para *deadlocks* nunca ocorra (1)
- **Exclusão mútua**
  - não há como evitar: condição intrinsecamente necessária para recursos não compartilháveis (ex.: impressora)

→ OBS.: não se aplica para recursos compartilháveis (ex.: arquivo somente leitura), pois não geram deadlock

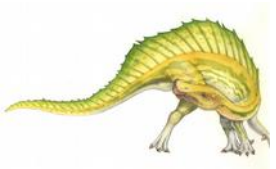


Prevenção de *deadlocks*

- Garanta que pelo menos uma das condições originais para *deadlocks* nunca ocorra (2)
- **Posse durante a espera**
  - não permita que processos peçam recursos “aos poucos”
    - A) pedem todos de uma vez ou
    - B) liberam todos os que detêm antes de pedir outros
  - pode levar à baixa utilização dos recursos ou inanição
    - ex.: *copie do DVD para arquivo no HD, classifique o arquivo e imprima os resultados na impressora*
      - A) mantém a impressora apesar de precisar dela só no final, mantém o DVD apesar de precisar só no início
      - B) pode demorar a conseguir o HD pela 2ª vez
        - 1ª vez DVD(r)→HD(w) ; 2ª vez HD(r)→IMPR(w)

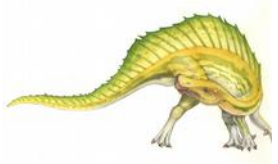
Prevenção de *deadlocks*

- Garanta que pelo menos uma das condições originais para *deadlocks* nunca ocorra (3)
- **Inexistência de preempção**
  - se um processo não conseguir alocar imediatamente um novo recurso, deve abrir mão dos que já detém (intercepta)
    - implicitamente, eles serão adicionados à sua lista de espera
    - o processo só poderá continuar quando todos os recursos puderem ser obtidos (antigos e novos)
  - pode levar à inanição



Prevenção de *deadlocks*

- Garanta que pelo menos uma das condições originais para *deadlocks* nunca ocorra (4)
- **Espera circular**
  - Defina uma ordenação absoluta para todos os tipos de recursos disponíveis
  - Exija que todo processo requisite recursos sempre em ordem crescente
    - ex.: DVD(1), HD(5), IMPR(12)
      - DVD+IMPR: deve solicitar primeiro o DVD e depois a IMPR
  - Impede a formação de ciclos no grafo, desde que o programador adquira os recursos na ordem apropriada

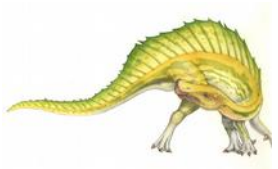


“Impedimento” de *deadlocks*

- No método mais simples e útil:
  - o S.O. recebe antecipadamente informações adicionais
- Cada processo declara o máximo de recursos de que pode vir a necessitar
- O algoritmo avalia dinamicamente cada alocação de recursos
  - visa garantir que não se forme nenhuma espera circular

Estado seguro (de segurança)

- Cada requisição é avaliada contra a alocação corrente e as demandas máximas declaradas
  - Há disponibilidade? Aloca (com segurança)!
  - Não há? Seria inseguro alocar!
- O sistema está em um estado seguro se existe uma sequência de alocação segura para todos
  - Sequência segura: $<P_j, P_i, \ldots, P_n>$
    - Processo P_j: Requisita → Aloca → Devolve
    - Processo P_i: Requisita → Aloca → Devolve
    - …
    - Processo P_n: Requisita → Aloca → Devolve



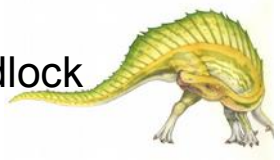
Estado seguro (de segurança)

- Exemplo: 12 acionadores de fitas, processos P0, P1, P2:

	P0	P1	P2
Necessidade máxima	10	4	9
Necessidade atual	5	2	2

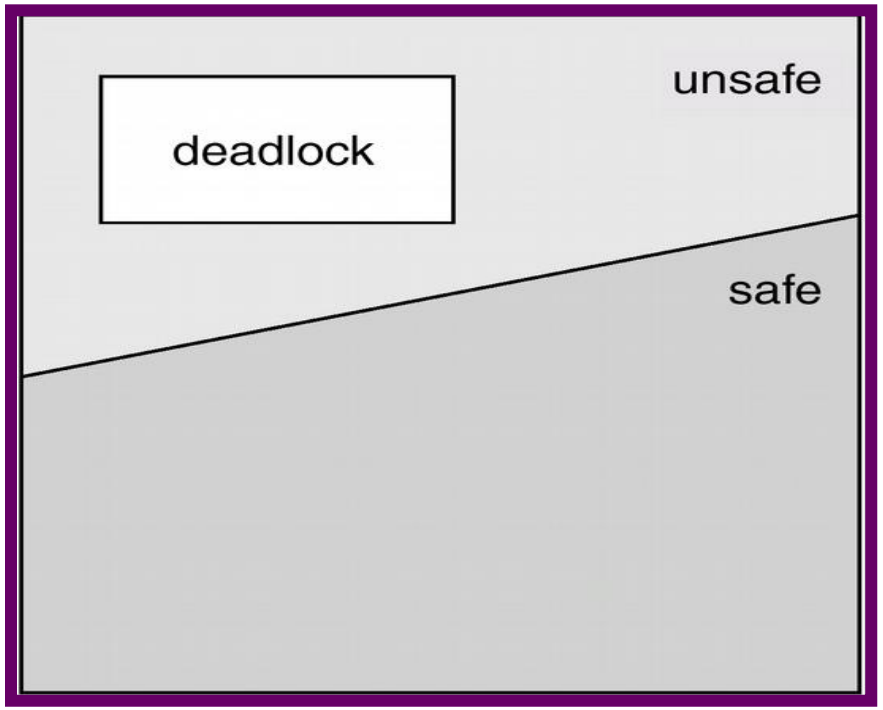
  - Atualmente, há somente 3 acionadores disponíveis:
    - 12 – (5+2+2)
  - **Estado seguro**: < P1 , P0 , P2 >
    - P1 pode pedir no máximo mais 2 <= 3
    - P0 pode pedir no máximo mais 5 <= 3 + 2
    - P2 pode pedir no máximo mais 7 <= 3 + 2 + 5

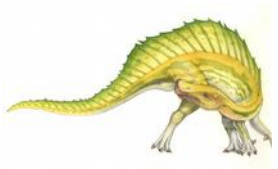
- O que aconteceria se o S.O. alocasse P0 ou P2 antes de P1?
  - <P0, ...> ou <P2, ...> são estados inseguros e que podem gerar deadlock



Conceitos básicos

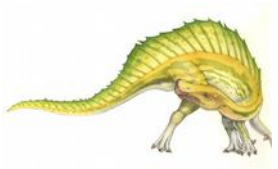
- Se um estado é seguro
  - Não há *deadlocks*
- Se um estado é inseguro
  - Há *possibilidade* de *deadlocks*
- Impedimento:
  - Garantir que o sistema nunca entre um estado inseguro





Algoritmo do banqueiro

- Aplicável a recursos com múltiplas cópias
- Cada processo deve indicar requisitos máximos
- Quem requisita algo pode ter que esperar
- Quem recebe recursos deve devolvê-los em um tempo finito



Algoritmo do banqueiro: exemplo

- Processos P0 – P4; recursos A (10), B(5), C(7)

	Aloc.			Max			Neces.		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Dispon.: A(3) B(3) C(2)

Seguro: <P1,P3,P4,P2,P0>

Algoritmo do banqueiro: exemplo

- Processos P0 – P4; recursos A (10), B(5), C(7)

	Aloc.			Max			Neces.		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Dispon.: A(3) B(3) C(2)

Seguro: <P1,P3,P4,P2,P0>

Algoritmo do banqueiro: exemplo

- Processos P0 – P4; recursos A (10), B(5), C(7)

	Aloc.			Max			Neces.		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Dispon.: A(5) B(3) C(2)

Seguro: <P1,P3,P4,P2,P0>

Algoritmo do banqueiro: exemplo

- Processos P0 – P4; recursos A (10), B(5), C(7)

	Aloc.			Max			Neces.		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Dispon.: A(7) B(4) C(3)

Seguro: <P1,P3,P4,P2,P0>

Algoritmo do banqueiro: exemplo

- Processos P0 – P4; recursos A (10), B(5), C(7)

	Aloc.			Max			Neces.		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Dispon.: A(7) B(4) C(5)

Seguro: <P1,P3,P4,P2,P0>

Algoritmo do banqueiro: exemplo

- Processos P0 – P4; recursos A (10), B(5), C(7)

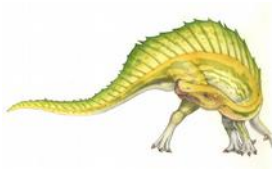
	Aloc.			Max			Neces.		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Dispon.: A(10) B(4) C(7)

Seguro: <P1,P3,P4,P2,P0>

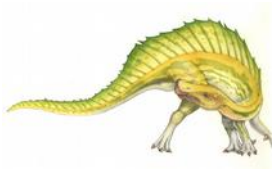
Detecção de *deadlocks*

- Permita ao sistema entrar em deadlock
- Implementa um algoritmo de detecção
  - Semelhante aos de impedimento
- Defina um esquema de recuperação



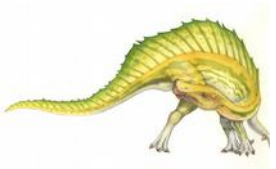
Aplicação do algoritmo de detecção

- Considerações a serem feitas:
  - Quão frequentes são os *deadlocks*?
  - Quantos processos são afetados?
- Detecção frequente:
  - Pouco tempo de espera
  - Pouca chance de “propagação” do travamento
- Detecção esporádica:
  - Menor *overhead* de detecção
  - Pode encontrar muitos ciclos



Recuperação de *deadlocks*

- É preciso quebrar os ciclos no grafo:
  - Abortando um ou mais processos
    - Processo termina com erro
    - Estado do sistema pode ficar inconsistente
  - Fazer a preempção de recursos
    - Processos que sofrem preempção precisam "retroceder" (*roll-back*) para um ponto anterior
- Considerações:
  - Como escolher o(s) processo(s) vítima(s)?
  - Como distribuir os recursos reclamados?
  - Como evitar a inanição?



Fim do Capítulo 7