Sistemas Operacionais

Deadlocks (cap. 8)

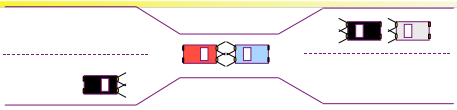
Sumário

- Modelo de sistema considerado
- *Deadlocks*
- Métodos para lidar com *deadlocks*:
  - prevenção (*prevention*)
  - impedimento (*avoidance*)
  - 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
- Exemplos:
  - Um sistema com duas unidades de fita
    - P1 e P2 já reservaram uma unidade e precisam da outra
  - 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 R1, R2, ..., Rm
  - Ciclos de CPU, espaço de memória, disp. E/S
- Cada recurso tem Ri tem Wi instâncias
- Processos acessam recursos da mesma forma:
  - requisita
  - utiliza
  - libera

Deadlock: condições necessárias

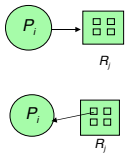
- Exclusão mútua
- Posse durante a espera
- Não preempção
- Espera circular

Grafo de alocação de recursos

- Vértices são divididos em dois tipos:
  - P = {P1, P2, …, Pn}, os processos no sistema
  - R = {R1, R2, …, Rm}, 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

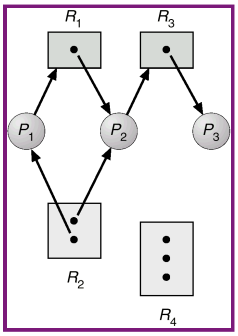
- Um processo:
- Tipo de recurso com 4 instâncias
- Pi requisita instância de Rj
- Pi detém uma instância de Rj



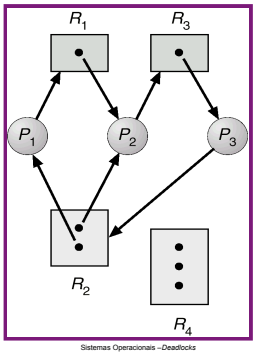
Fatos básicos

- Se não há ciclos no grafo → não há *deadlock*
- Se o grafo contém ciclos
  - Se recursos só têm uma instância → *deadlock*
  - Se há mais de uma instância → possível *deadlock*

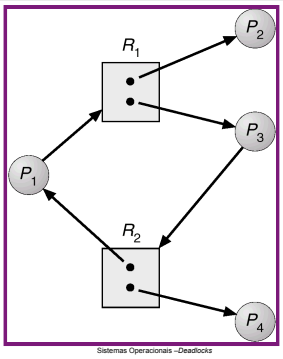
Grafo de alocação de recursos



Grafo de alocação de recursos com *deadlock*



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



Formas de lidar com *deadlocks*

- Garanta que por construção eles não acontecem
- Evite *deadlocks* antes que eles ocorram
- Detecte quando um *deadlock* ocorre e recupere o sistema a um estado aceitável
- Ignore o problema e faça de conta que eles nunca acontecem
  - Usado na maioria dos sistemas, inclusive o Linux!

Prevenção de *deadlocks*

- Garanta que pelo menos uma das condições originais para *deadlocks* nunca ocorra
  - Exclusão mútua
  - Posse durante a espera
  - Não preempção
  - Espera circular

Prevenção de *deadlocks*

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

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:
    - pedem todos de uma vez ou
    - liberam todos os que detêm antes de pedir outros
  - Pode levar à baixa utilização dos recursos ou inanição

Prevenção de *deadlocks*

- Garanta que pelo menos uma das condições originais para *deadlocks* nunca ocorra (3)
- Não preempção
  - Se um processo não conseguir alocar um novo recurso, deve abrir mão dos que já detém
    - Implicitamente, eles serão adicionados à sua lista de requisições atual
    - O processo só poderá continuar quando todos os recursos puderem ser obtidos
  - 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 total para todos os tipos de recursos disponíveis
  - Exija que todo processo requisite recursos sempre em ordem crescente
    - Impede a formação de ciclos no grafo

"Impedimento" de *deadlocks*

- Método mais simples e útil
- Cada processo declara o máximo de recursos de que pode vir a necessitar
- O algoritmo avalia dinamicamente cada alocação de recursos para garantir que não se forme nenhuma espera circular
- Alocação de recursos é feita pela disponibilidade instantânea e as demandas máximas declaradas
  - Técnica adotada, p.ex., em redes ATM

Estado seguro (de segurança)

- Cada requisição é avaliada contra a alocação corrente e as demandas máximas declaradas
- O sistema está em um estado seguro se existe uma sequência de alocação segura para todos

Estado seguro (de segurança)

- Uma seq. <P1,P2,...,Pn> é segura se para Pi,
  - os recursos que Pi pode requisitar não excedem a soma do que está disponível com as demandas máximas de todos os Pj, j<i
  - se recursos não estiverem disponíveis, Pi pode esperar pelo término de todos os Pj
  - quando Pi terminar, ele liberará todos os recursos, que podem ser usados por Pi+1

Estado seguro (de segurança)

- Exemplo: 12 acionadores de fitas, 3 processos

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

  - Estado seguro: <P1,P0,P2>
    - Há 12 – (5+2+2) = 3 acionadores disponíveis
    - P1 pode pedir no máximo mais 2
    - P0 pode pedir no máximo 5 = 3 + 2
    - P2 pode pedir no máximo 7 < 5 + 3 + 2
  - O que acontece se P2 requisitar (e receber) mais um acionador?

Estado seguro (de segurança)

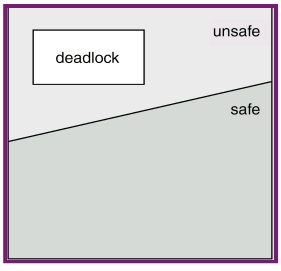
- Exemplo: 12 acionadores de fitas, 3 processos

	P0	P1	P2
Necessidade máxima	10	4	9
Necessidade atual	5	2	3

  - Estado inseguro
    - Há apenas dois acionadores disponíveis
      - Isso não atende a demanda máxima de P0, nem de P2
    - P1 só pode pedir mais 2, então eventualmente termina
    - Restariam 4 acionadores livres, que não atendem as demandas máximas de P0 nem de P2

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



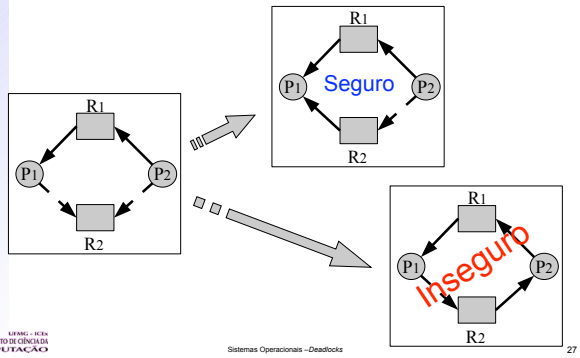
Algoritmo do grafo de alocação de recursos

- Válido para recursos únicos (sem réplicas)
- Aresta de requisição Pi → Rj (tracejada)
  - Processo Pi pode vir a requisitar Rj
  - Torna-se aresta de solicitação quando o pedido ocorre realmente
  - Quando o recurso é liberado aresta volta a ser de requisição
- Todos os recursos devem ser requisitados *a priori* no sistema

Algoritmo do grafo de alocação de recursos

- Antes de atender qualquer requisição:
  - Verifica-se se ao atender a req. (inverter a aresta de requisição) cria-se um ciclo
  - Se o ciclo existe, estado seria inseguro
    - Processo é suspenso temporariamente
  - Se o ciclo não existe, estado é seguro
    - Solicitação pode ser atendida

Algoritmo do grafo de alocação de recursos



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: estruturas de dados necessárias

- Sejam n processos e m tipos de recursos:
  - Disponível[m]: no. de instâncias disponíveis
  - Max[n][m]: demandas máximas de cada processo
  - Alocação[n][m]: situação corrente
  - Necessidade[n][m]: quanto ainda se pode pedir
    - Need [i,j] = Max[i,j] – Allocation [i,j]

Algoritmo do banqueiro: teste de segurança

Sejam Trabalho[m] = Disponível,
Término[n] = { false, ...}

0) Ache um i (processo) tal que
   a) Término[i] == false
   b) Necessidade[i] <= Trabalho

$O(m \times n^2)$

1) Trabalho = Trabalho + Alocação[i]
   Término[i] = true
   Retorne ao passo 1

0) Seguro = (Término[i] para todo i)

Algoritmo do banqueiro: solicitação de recursos por Pi

Solicitação[m] = demandas de Pi.

0) Se Solicitação > Necessidade → ERRO
1) Se Solicitação > Disponível → Pi aguarda
2) Caso contrário, aloque os recursos:
   a) Disponível = Disponível – Solicitação
   b) Alocação[i] = Alocação[i] + Solicitação
   c) Necessidade[i] = Necessidade[i] – Solicitação
3) Se estado final é seguro → FIM
4) Senão, restaura estado anterior e Pi espera

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>

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)

P1 requisita (1 0 2) → aceitar ou não?

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	3 0 2	3 2 2	0 2 0
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(2) B(3) C(0)

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

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	3 0 2	3 2 2	0 2 0
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,P0,P2>

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	3 0 2	3 2 2	0 2 0
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,P0,P2>

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	3 0 2	3 2 2	0 2 0
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,P0,P2>

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	3 0 2	3 2 2	0 2 0
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(5) C(5)

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

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	3 0 2	3 2 2	0 2 0
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(2) B(3) C(0)

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

P0 requisita (3 3 0) → aceitar ou não?

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	3 0 2	3 2 2	0 2 0
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(2) B(3) C(0)

Não há recursos suficientes p/ (3 3 0)

P0 requisita (0 2 0) → aceitar ou não?

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 3 0	7 5 3	7 2 3
P1	3 0 2	3 2 2	0 2 0
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(2) B(1) C(0)

Estado inseguro

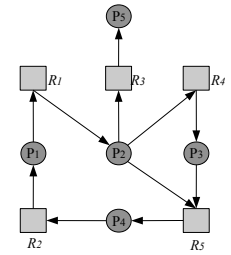
Detecção de *deadlocks*

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

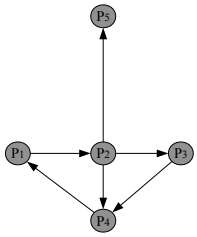
Detecção de *deadlock* quando recursos têm instâncias únicas

- Manter um grafo de espera (“espere por”)
  - derivado do grafo de alocação
  - vértices: apenas processos
  - Pi → Pj: Pi solicita recurso detido por Pj
- Deadlock: ciclo no grafo (espera circular)
- Periodicamente executa algoritmo de busca de ciclos ($O(n^2)$)

Montagem do grafo "espere-por"



Grafo de alocação de recursos



Grafo "espere-por" correspondente

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Algoritmo semelhante à detecção de estado seguro no algoritmo do banqueiro:
  - Disponível[m]: no. de instâncias disponíveis
  - Alocação[n][m]: situação corrente
  - Solicitação[n][m]: quanto cada processo pede

Detecção de *deadlock* quando recursos têm múltiplas instâncias

Sejam Trabalho[m] = Disponível,
Término[n] = { soma(alocação[n][ ]) == 0 }

0) Ache um i (processo) tal que
   a) Término[i] == false
   b) Solicitação[i] <= Trabalho

O(m x n²)

1) Trabalho = Trabalho + Alocação[i]
   Término[i] = true
   Retorne ao passo 1

0) Deadlock = (Término[i] = false para algum i)

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.	Solic.
	A B C	A B C
P0	0 1 0	0 0 0
P1	2 0 0	2 0 2
P2	3 0 3	0 0 0
P3	2 1 1	1 0 0
P4	0 0 2	0 0 2

Dispon.: A(0) B(0) C(0)
Sem *deadlock*: <P0,P2,P3,P1,P4>

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.	Solic.
	A B C	A B C
P0	0 1 0	0 0 0
P1	2 0 0	2 0 2
P2	3 0 3	0 0 0
P3	2 1 1	1 0 0
P4	0 0 2	0 0 2

Dispon.: A(0) B(0) C(0)
Sem *deadlock*: <P0,P2,P3,P1,P4>

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.	Solic.
	A B C	A B C
P0	0 1 0	0 0 0
P1	2 0 0	2 0 2
P2	3 0 3	0 0 0
P3	2 1 1	1 0 0
P4	0 0 2	0 0 2

Dispon.: A(3) B(1) C(3)
Sem *deadlock*: <P0,P2,P3,P1,P4>

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.			Solic.		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Dispon.: A(5) B(2) C(4)

Sem *deadlock*: <P0,P2,P3,P1,P4>

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.			Solic.		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Dispon.: A(7) B(2) C(4)

Sem *deadlock*: <P0,P2,P3,P1,P4>

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.			Solic.		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Dispon.: A(0) B(0) C(0)

Agora, suponhamos que P2 solicita C(1)

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.			Solic.		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	1
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Dispon.: A(0) B(0) C(0)

Deadlock!

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.			Solic.		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	1
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Dispon.: A(0) B(0) C(0)

Deadlock!

Detecção de *deadlock* quando recursos têm múltiplas instâncias

- Processos P0 – P4; recursos A (7), B(2), C(6)

	Aloc.			Solic.		
	A	B	C	A	B	C
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	1
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Dispon.: A(0) B(1) C(0)

Deadlock!

Aplicação do algoritmo de detecção

- 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
- Abortar 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

Recuperação de *deadlocks*: questões

- Como escolher o(s) processo(s) vítima(s)?
- Como distribuir os recursos reclamados?
- Como evitar a inanição?