

UFFS

Ciência da Computação

Sistemas Operacionais

Prof. Marco Aurélio Spohn

Deadlocks - Impasses

- 3.1. Recurso
- 3.2. Introdução aos *deadlocks*
- 3.3. Algoritmo do avestruz
- 3.4. Detecção e recuperação de *deadlocks*
- 3.5. Evitando *deadlocks*
- 3.6. Prevenção de *deadlocks*
- 3.7. Outras questões

Recursos

- Exemplos de recursos de computador
 - impressoras
 - unidades de fita
 - tabelas
- Processos precisam de acesso aos recursos numa ordem racional
- Suponha que um processo detenha o recurso A e solicite o recurso B
 - ao mesmo tempo um outro processo detém B e solicita A
 - ambos são bloqueados e assim permanecem

Recursos (1)

- *Deadlocks* ocorrem quando ...
 - garante-se aos processos acesso exclusivo aos dispositivos
 - esses dispositivos são normalmente chamados de recursos
- Recursos preemptíveis
 - podem ser retirados de um processo sem quaisquer efeitos prejudiciais
- Recursos não preemptíveis
 - vão induzir o processo a falhar se forem retirados

Recursos (2)

- Seqüência de eventos necessários ao uso de um recurso
 - 1.solicitar o recurso
 - 2.usar o recurso
 - 3.liberar o recurso
- Deve esperar se solicitação é negada
 - processo solicitante pode ser bloqueado
 - pode falhar resultando em um código de erro

Introdução aos *Deadlocks*

Definição formal:

- Um conjunto de processos está em situação de *deadlock* se todo processo pertencente ao conjunto estiver esperando por um evento que somente um outro processo desse mesmo conjunto poderá fazer acontecer
- Normalmente o evento é a liberação de um recurso atualmente retido
- Nenhum dos processos pode...
 - executar
 - liberar recursos
 - ser acordado

Quatro Condições para *Deadlock*

1. Condição de exclusão mútua

- todo recurso está associado a um processo ou está disponível

2. Condição de posse e espera

- processos que retêm recursos podem solicitar novos recursos

3. Condição de não preempção

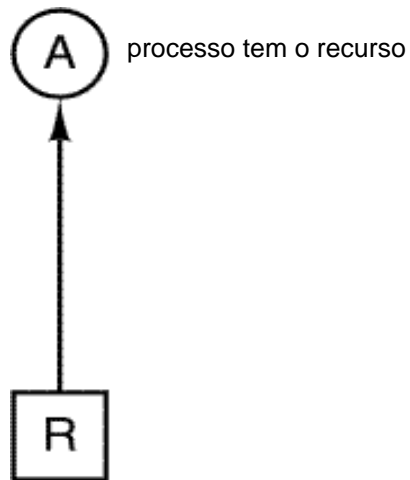
- recursos concedidos previamente não podem ser forçosamente tomados

4. Condição de espera circular

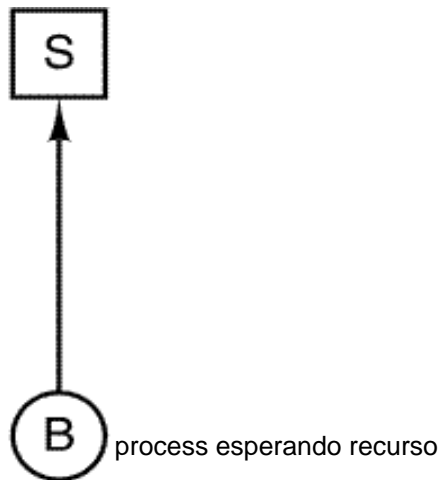
- deve ser uma cadeia circular de 2 ou mais processos
- cada um está à espera de recurso retido pelo membro seguinte dessa cadeia

Modelagem de *Deadlock* (2)

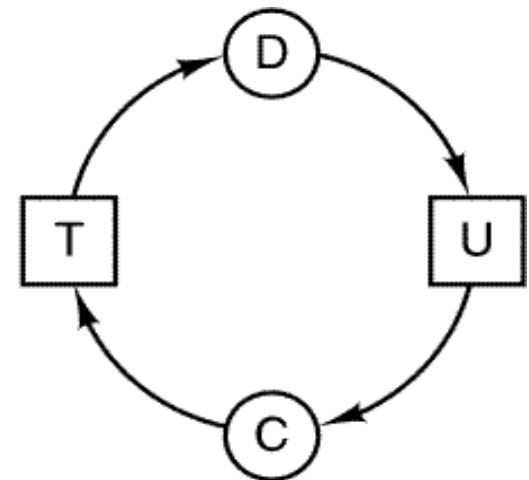
- Modelado com grafos dirigidos



(a)



(b)



(c)

a) recurso R alocado ao processo A

b) processo B está solicitando/esperando pelo recurso S

c) processos C e D estão em *deadlock* sobre recursos T e U

Modelagem de *Deadlock* (3)

- Estratégias frente a *Deadlocks*

- 1.ignorar por completo o problema

- 2.deteccção e recuperação

- 3.anulação dinâmica (evita-se)

- **alocação cuidadosa** de recursos

- 4.prevenção (previne-se)

- **negação** de uma das quatro condições necessárias

Modelagem de *Deadlock* (4)

A
Requisita R
Requisita S
Libera R
Libera S

(a)

B
Requisita S
Requisita T
Libera S
Libera T

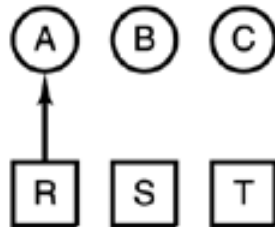
(b)

C
Requisita T
Requisita R
Libera T
Libera R

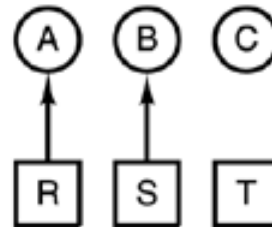
(c)

1. A requisita R
 2. B requisita S
 3. C requisita T
 4. A requisita S
 5. B requisita T
 6. C requisita R
- deadlock

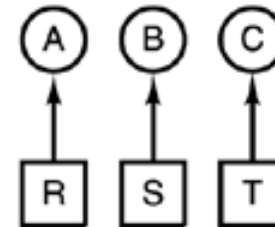
(d)



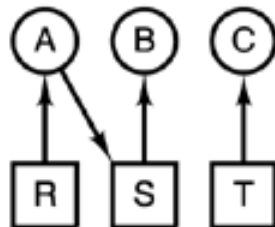
(e)



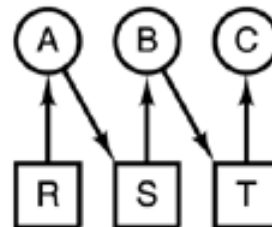
(f)



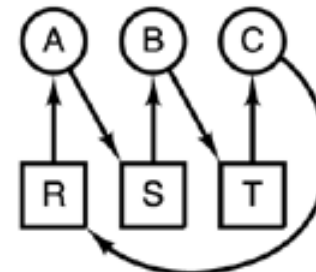
(g)



(h)



(i)



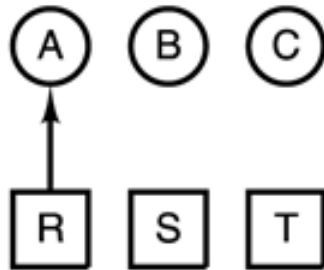
(j)

- Como ocorre um *deadlock*

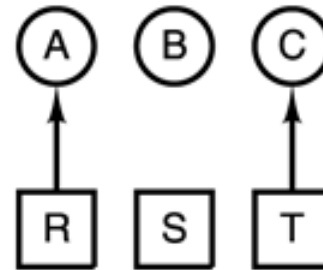
Modelagem de *Deadlock* (5)

1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock

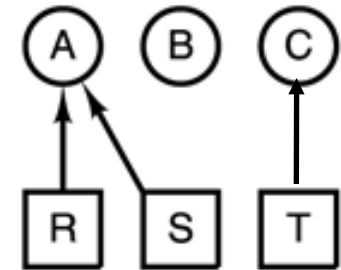
(k)



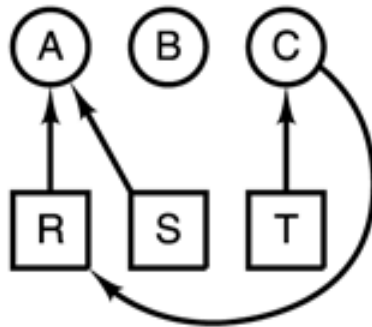
(l)



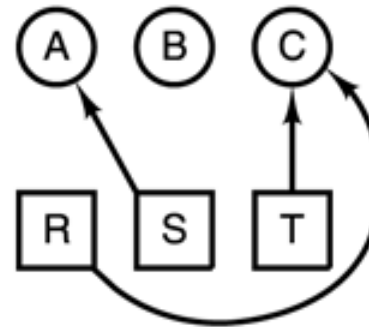
(m)



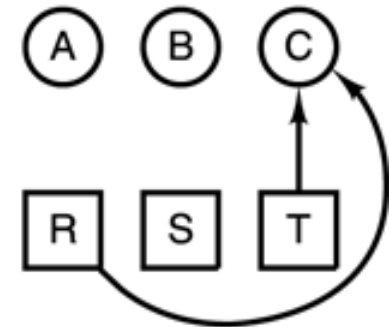
(n)



(o)



(p)



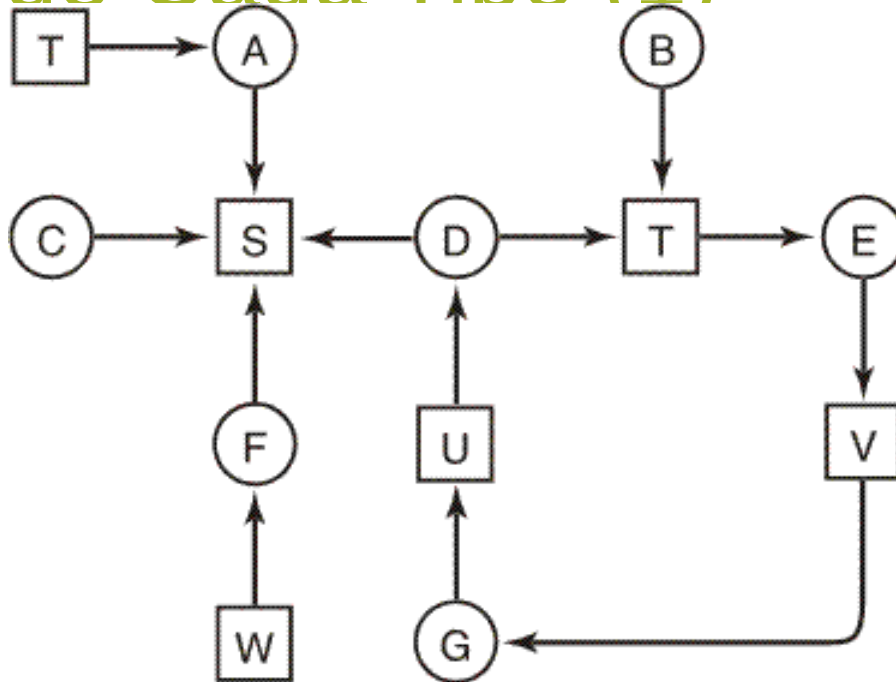
(q)

- Como pode ser evitado um *deadlock*

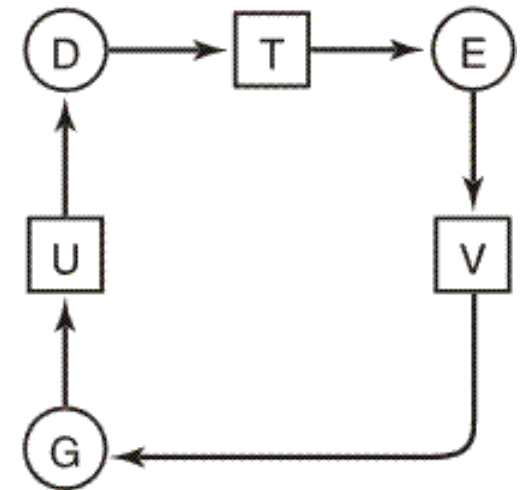
Algoritmo do Avestruz

- Finge que o problema não existe
- Razoável se
 - *deadlocks* ocorrem muito raramente
 - custo da prevenção é alto
- **UNIX e Windows seguem esta abordagem**
- É uma ponderação entre
 - conveniência
 - correção

Detecção com um Recurso de Cada Tipo (1)



(a)




(b)

- (a) Observe a posse e solicitações de recursos
- (b) Um ciclo pode ser encontrado dentro do grafo, denotando *deadlock*

Detecção com múltiplos Recursos de Cada Tipo (2)

Recursos existentes
($E_1, E_2, E_3, \dots, E_m$)

Matriz de alocação atual




C_{11}	C_{12}	C_{13}	\dots	C_{1m}
C_{21}	C_{22}	C_{23}	\dots	C_{2m}
\vdots	\vdots	\vdots		\vdots
C_{n1}	C_{n2}	C_{n3}	\dots	C_{nm}

Linha n é a alocação
atual para o processo n

Recursos disponíveis
($A_1, A_2, A_3, \dots, A_m$)

Matriz de requisições



R_{11}	R_{12}	R_{13}	\dots	R_{1m}
R_{21}	R_{22}	R_{23}	\dots	R_{2m}
\vdots	\vdots	\vdots		\vdots
R_{n1}	R_{n2}	R_{n3}	\dots	R_{nm}

Linha 2 informa qual é a
necessidade do processo 2

- Estruturas de dados necessárias ao algoritmo de detecção de *deadlock*

Detecção com múltiplos Recursos de Cada Tipo (?)

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Unidades de fita
Plotters
Scanners
Unidades de CD-ROM

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Unidades de fita
Plotters
Scanners
Unidades de CD-ROM

Matriz alocação atual

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Matriz de requisições

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Um exemplo para o algoritmo de detecção de deadlocks

Recuperação de Deadlock (1)

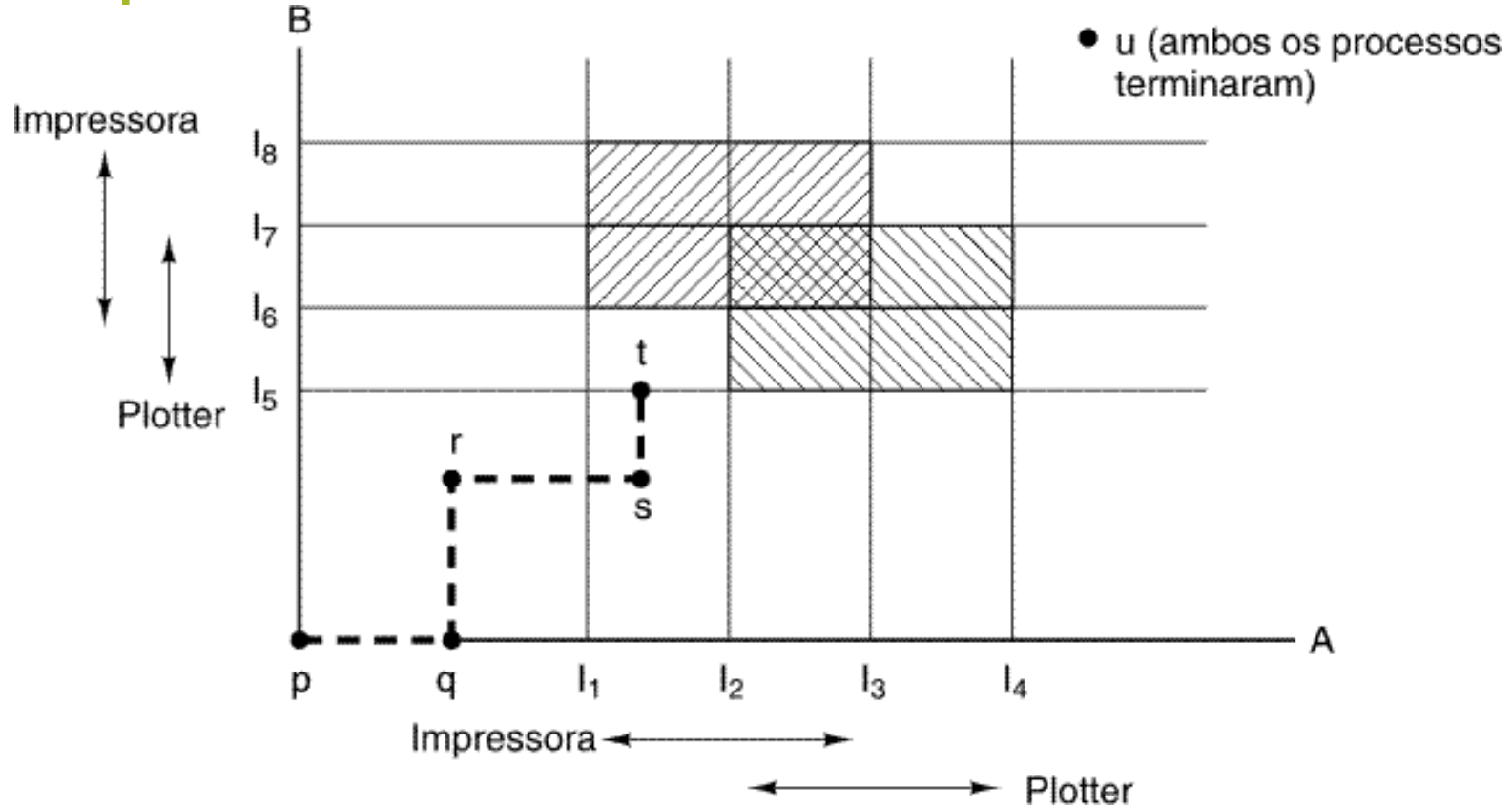
- Recuperação através de preempção
 - retirar um recurso de algum outro processo
 - depende da natureza do recurso
- Recuperação através de reversão de estado
 - verifica um processo periodicamente
 - usa este estado salvo
 - reinicia o processo se este é encontrado em estado de *deadlock*

Recuperação de Deadlock (2)

- Recuperação através da eliminação de processos
 - forma mais grosseira mas também mais simples de quebrar um *deadlock*
 - elimina um dos processos no ciclo de *deadlock*
 - os outros processos conseguem seus recursos
 - **escolhe processo que pode ser reexecutado desde seu início**

Evitando Deadlocks

Trajeto rias de Recursos



- Trajet rias de recursos de dois processos (um(a) s  CPU/n cleo): **algumas regi es s o inaceit veis** (exclus o m tua: regi es com preenchimento em diagonal representam viola  o da exclus o m tua); nesse exemplo, primeiro A executa por completo e depois B executa, ou vice-versa (ambos finalizam com sucesso, atingindo o ponto u de finaliza  o).

Estados Seguros e Inseguros (1)

Possui máx.	Possui máx.	Possui máx.	Possui máx.	Possui máx.																																													
<table><tr><td>A</td><td>3</td><td>9</td></tr><tr><td>B</td><td>2</td><td>4</td></tr><tr><td>C</td><td>2</td><td>7</td></tr></table>	A	3	9	B	2	4	C	2	7	<table><tr><td>A</td><td>3</td><td>9</td></tr><tr><td>B</td><td>4</td><td>4</td></tr><tr><td>C</td><td>2</td><td>7</td></tr></table>	A	3	9	B	4	4	C	2	7	<table><tr><td>A</td><td>3</td><td>9</td></tr><tr><td>B</td><td>0</td><td>–</td></tr><tr><td>C</td><td>2</td><td>7</td></tr></table>	A	3	9	B	0	–	C	2	7	<table><tr><td>A</td><td>3</td><td>9</td></tr><tr><td>B</td><td>0</td><td>–</td></tr><tr><td>C</td><td>7</td><td>7</td></tr></table>	A	3	9	B	0	–	C	7	7	<table><tr><td>A</td><td>3</td><td>9</td></tr><tr><td>B</td><td>0</td><td>–</td></tr><tr><td>C</td><td>0</td><td>–</td></tr></table>	A	3	9	B	0	–	C	0	–
A	3	9																																															
B	2	4																																															
C	2	7																																															
A	3	9																																															
B	4	4																																															
C	2	7																																															
A	3	9																																															
B	0	–																																															
C	2	7																																															
A	3	9																																															
B	0	–																																															
C	7	7																																															
A	3	9																																															
B	0	–																																															
C	0	–																																															
Disponível: 3	Disponível: 1	Disponível: 5	Disponível: 0	Disponível: 7																																													
(a)	(b)	(c)	(d)	(e)																																													

- Demonstração de que o estado em (a) é seguro

Estados Seguros e Inseguros (2)

Possui máx.

A	3	9
B	2	4
C	2	7

Disponível: 3

(a)

Possui máx.

A	4	9
B	2	4
C	2	7

Disponível: 2

(b)

Possui máx.

A	4	9
B	4	4
C	2	7

Disponível: 0

(c)

Possui máx.

A	4	9
B	–	–
C	2	7

Disponível: 4

(d)

- Demonstração de que o estado em (b) é inseguro

O Algoritmo do Banqueiro para um Único Recurso (by Dijkstra):

verifica se há uma sequência de atendimento que permita atender a todos os clientes

Possui máx.

A	0	6
B	0	5
C	0	4
D	0	7

Disponível: 10

(a)

Possui máx.

A	1	6
B	1	5
C	2	4
D	4	7

Disponível: 2

(b)

Possui máx.

A	1	6
B	2	5
C	2	4
D	4	7

Disponível: 1

(c)

- Três estados de alocação de recursos
 - seguro
 - seguro
 - inseguro

O Algoritmo do Banqueiro para Múltiplos Recursos

	Processo	Unidades de fita	Plotters	Scanners	Unidades de CD-ROM
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	
Recursos alocados					
	Processo	Unidades de fita	Plotters	Scanners	Unidades de CD-ROM
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	
Recursos ainda necessários					

$E = (6342)$
 $P = (5322)$
 $A = (1020)$

- Exemplo do algoritmo do banqueiro com múltiplos recursos

Prevenção de *Deadlock*

Atacando a Condição de Exclusão Mútua

- Alguns dispositivos (como uma impressora) podem fazer uso de *spool*
 - o **daemon** de impressão é o único que usa o recurso impressora
 - desta forma **deadlock** envolvendo a impressora é eliminado
- Nem todos os dispositivos podem fazer uso de *spool*
- Princípio:
 - evitar alocar um recurso quando ele não for absolutamente necessário
 - tentar assegurar que o menor número possível de processos possa de fato requisitar o recurso

Prevenção de Deadlock

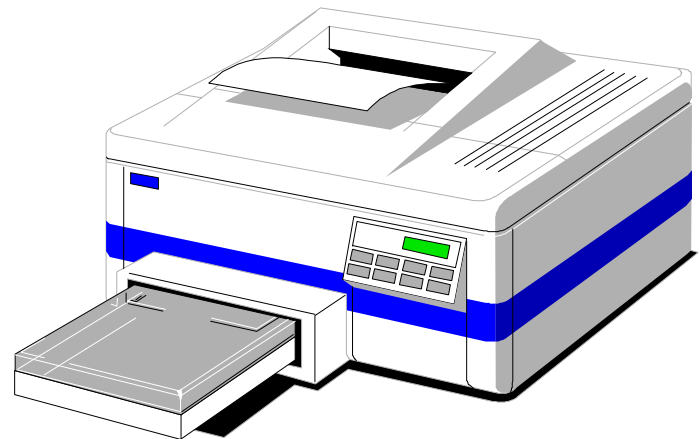
Atacando a Condição de Posse e Espera

- Exigir que todos os processos requisitem os recursos antes de iniciarem
 - um processo nunca tem que esperar por aquilo que precisa
- Problemas
 - podem não saber quantos e quais recursos vão precisar no início da execução
 - e também retêm recursos que outros processos poderiam estar usando (*sobreprovisionamento/overprovisioning*)
- Variação:
 - processo deve desistir de todos os recursos já adquiridos para então requisitar todos os que são imediatamente necessários

Prevenção de Deadlock

Atacando a Condição de Não Preempção

- Esta é, geralmente, uma opção inviável
- Considere um processo de posse de uma impressora
 - no meio da impressão, retoma-se a impressora à força
 - Que tal!!??

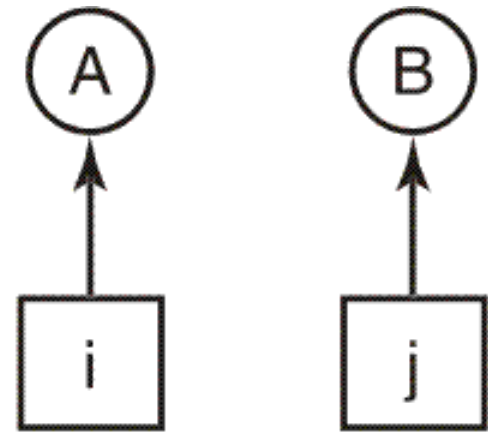


Prevenção de Deadlock

Atacando a Condição de Espera Circular

1. Imagesetter
2. Scanner
3. Plotter
4. Unidade de fita
5. Unidade de CD-ROM

(a)



(b)

a) Recursos ordenados numericamente: recursos são adquiridos em ordem estritamente crescente

b) Um grafo de recursos

Prevenção de Deadlock

Condição	Abordagem contra deadlocks
Exclusão mútua	Usar spool em tudo
Posse-e-espera	Requisitar inicialmente todos os recursos necessários
Não preempção	Retomar os recursos alocados
Espera circular	Ordenar numericamente os recursos

- Resumo das abordagens para prevenir deadlock

Outras Questões

Bloqueio em Duas Fases / *Two Phase Lock* (fase 1: aquisição; fase 2: liberação)

- Fase um
 - processo tenta bloquear todos os registros de que precisa, um de cada vez
 - Se registro necessário já estiver bloqueado, reinicia novamente
 - **(nenhum trabalho real é feito na fase um)**
- Se a fase um for bem sucedida, começa a fase dois,
 - execução de atualizações
 - liberação de bloqueios
- Observe a similaridade com a requisição de todos os recursos de uma só vez
- Algoritmo funciona onde o programador tiver organizado tudo cuidadosamente para que
 - o programa possa ser parado, reiniciado

Deadlocks que **não** envolvem recursos (*i.e.*, dispositivos)

- É possível que dois processos entrem em situação de *deadlock*
 - cada um espera que o outro faça algo
- Pode ocorrer com semáforos
 - cada processo executa um *down()* sobre dois semáforos (*mutex* e outro qualquer)
 - se executados na ordem errada, resulta em *deadlock*

Deadlocks que não envolvem recursos (*i.e.*, dispositivos)

- Impasse de comunicação
- **Livelock**: situação equivalente a um *deadlock*, pois o(s) processo(s) envolvido(s) faz(em) uso de sua(s) parcela(s) de CPU, mas não fica(m) bloqueado(s), pois testam repetidamente por uma condição que nunca será satisfeita!

Condição de Inanição (*Starvation*)

- Algoritmo para alocar um recurso
 - pode ser ceder para o *job mais curto primeiro*
- Funciona bem para múltiplos *jobs* curtos em um sistema
- *Jobs* longos podem ser preteridos indefinidamente
 - mesmo não estando bloqueados
- solução:
 - política do *primeiro a chegar, primeiro a ser servido*