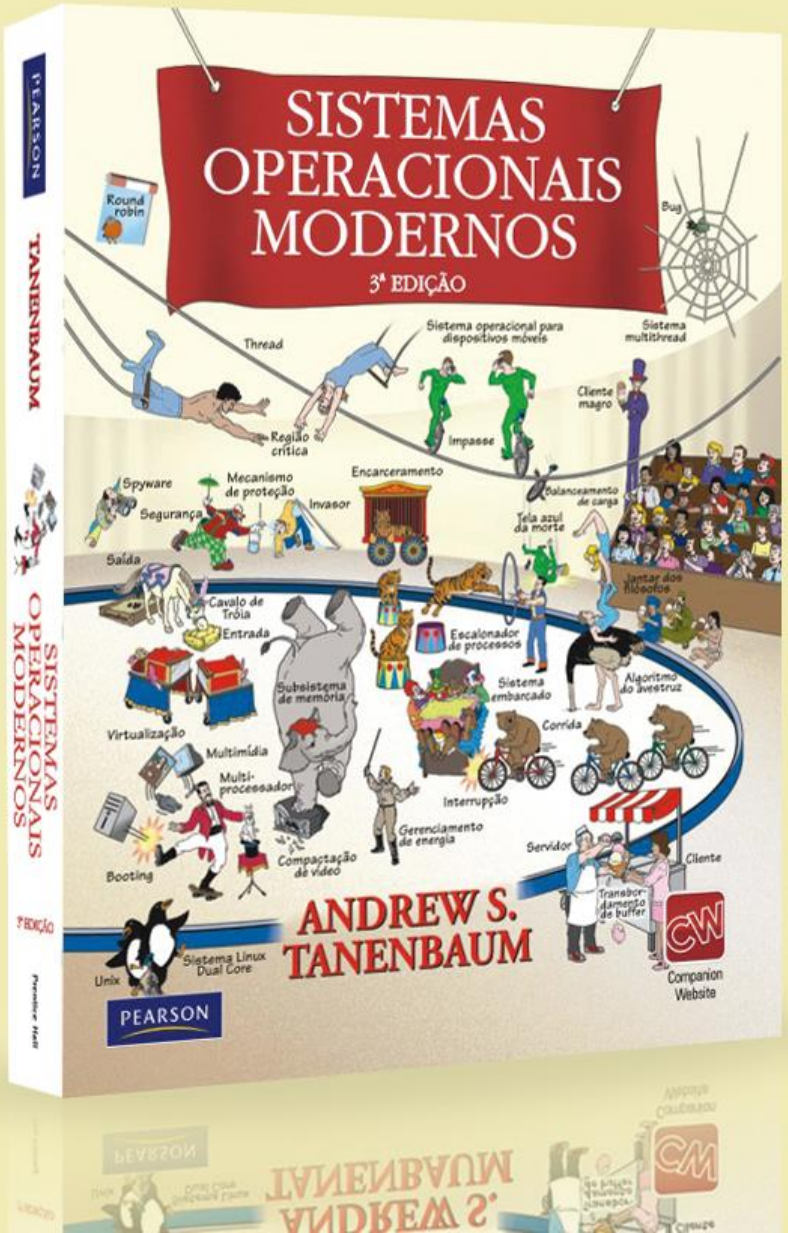
Sistemas operacionais modernos

Terceira edição

ANDREW S. TANENBAUM

Capítulo 6
Impasses



Impasses

Todos os SOs têm a capacidade de conceder (temporariamente), a um processo, acesso exclusivo a determinados recursos

Exemplo:

dois processos querem gravar um documento escaneado em um disco Blu-ray

- processo A solicita permissão para usar o scanner: SO concede recurso
- processo B solicita o gravador Blu-ray: SO concede recurso
- processo A pede pelo gravador Blu-ray

  $\Rightarrow$ solicitação é suspensa até que B libere o gravador

- antes de liberar o gravador Blu-ray, B pede pelo scanner

  $\Rightarrow$ **Ambos os processos estão bloqueados e assim permanecerão para sempre. Essa situação é chamada de impasse (*deadlock*).**

Recursos

“um recurso é algo que pode ser adquirido, usado e liberado com o passar do tempo”

“pode ser um dispositivo de hardware ou uma informação, por exemplo, um registro travado de uma base de dados”

Recursos preemptíveis e não preemptíveis

Preemptível: é aquele que pode ser retirado do processo proprietário sem nenhum prejuízo. **Ex: memória**

Não-preemptível: não pode ser retirado do atual processo sem que a causar prejuízos aos processos, causando falha na computação. **Ex: gravador de CD**

Normalmente, impasses envolvem recursos não-preemptíveis!

Recursos preemptíveis e não preemptíveis

Sequência de eventos necessária ao uso de um determinado recurso é dada abaixo de maneira abstrata:

1. Requisitar o recurso.
2. Usar o recurso.
3. Liberar o recurso.

Aquisição de recursos

A solicitação de recursos depende do tipo de SO:

- Chamada *Request* – em sistemas com chamadas explícitas para adquirir um recurso
- Chamada *Open* – em sistemas nos quais os recursos são arquivos especiais

✓ Cabe aos processos de usuário gerenciar o uso de recursos;

✓ Normalmente, associa-se um semáforo (mutex) a cada recurso;

✓ Operações *down e up* controlam o acesso aos recursos;

Introdução aos impasses

Definição:

Um conjunto de processos estará em situação de impasse se cada processo pertencente ao conjunto estiver esperando por um evento que somente outro processo desse mesmo conjunto poderá causar

Como todos os processos estão esperando, nenhum deles jamais causará qualquer evento que possa desbloquear/despertar um dos outros membros do conjunto, ***causando um impasse***

Introdução aos impasses

Considerações importantes:

- Processos tem um único thread;
- Não existem interrupções possíveis para acordar qualquer um dos processos bloqueados;

- ✓ "evento" que um processo está esperando é a liberação de um recurso que está de posse de um outro processo em situação de impasse

- ✓ nenhum dos processos pode continuar a execução, nenhum deles pode liberar qualquer recurso (HW ou SW) e nenhum deles pode ser acordado

⇒ **Impasse de recurso!**

Condições para impasse de recursos

1. **Condição de exclusão mútua** – **Recurso** está associado a um único processo ou está disponível

2. **Condição de posse e espera** – **Processos** que detêm algum recurso podem requisitar novos recursos

3. **Condição de não preempção** – **Recursos fornecidos** a um processo **não podem ser tomados** do mesmo. Têm que ser liberados pelo processo

4. **Condição de espera circular** – Deve existir um **encadeamento circular** de dois ou mais processos; cada um à espera de outro recurso que está sendo usado pelo membro da cadeia.

Modelagem de impasses

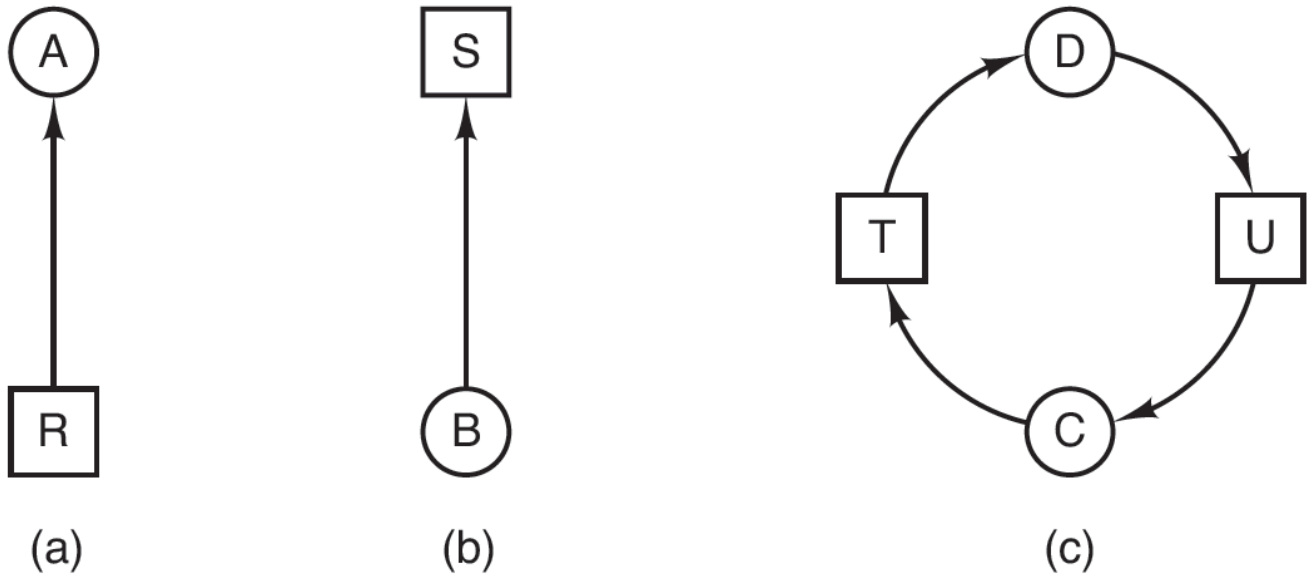


Figura 6.3 Grafos de alocação de recursos. (a) Processo de posse de um recurso. (b) Processo requisitando um recurso. (c) Impasse.

Modelagem de impasses

Um ciclo no grafo indica que existe um impasse que envolve os processos e recursos

Modelagem de impasses

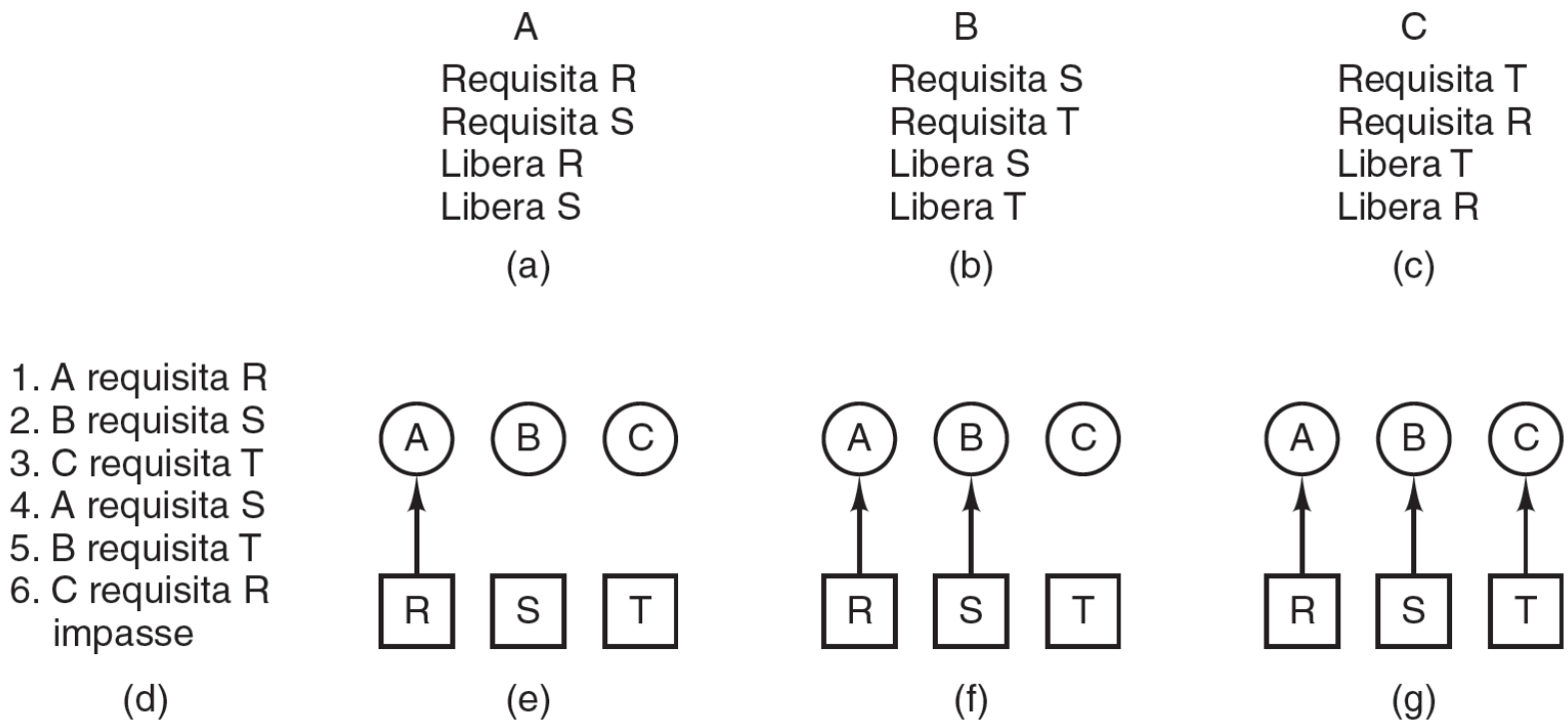


Figura 6.4 Exemplo de como um impasse ocorre e como pode ser evitado.

Modelagem de impasses

- ✓ O SO é livre para colocar em execução, em qualquer instante, qualquer processo que não esteja bloqueado

- ✓ Ex: A executa até o seu final, B idem, C idem

  $\Rightarrow$ não há impasses (*6.4a a 6.4c*)

- ✓ Processos sequenciais: desempenho cai (não há paralelismo), mas não há impasses

Modelagem de impasses

Alternância circular (round-robin) dos processos (6.4d):

Implica em figs de *6.4e a 6.4j* $\Rightarrow$ impasse!!

Modelagem de impasses

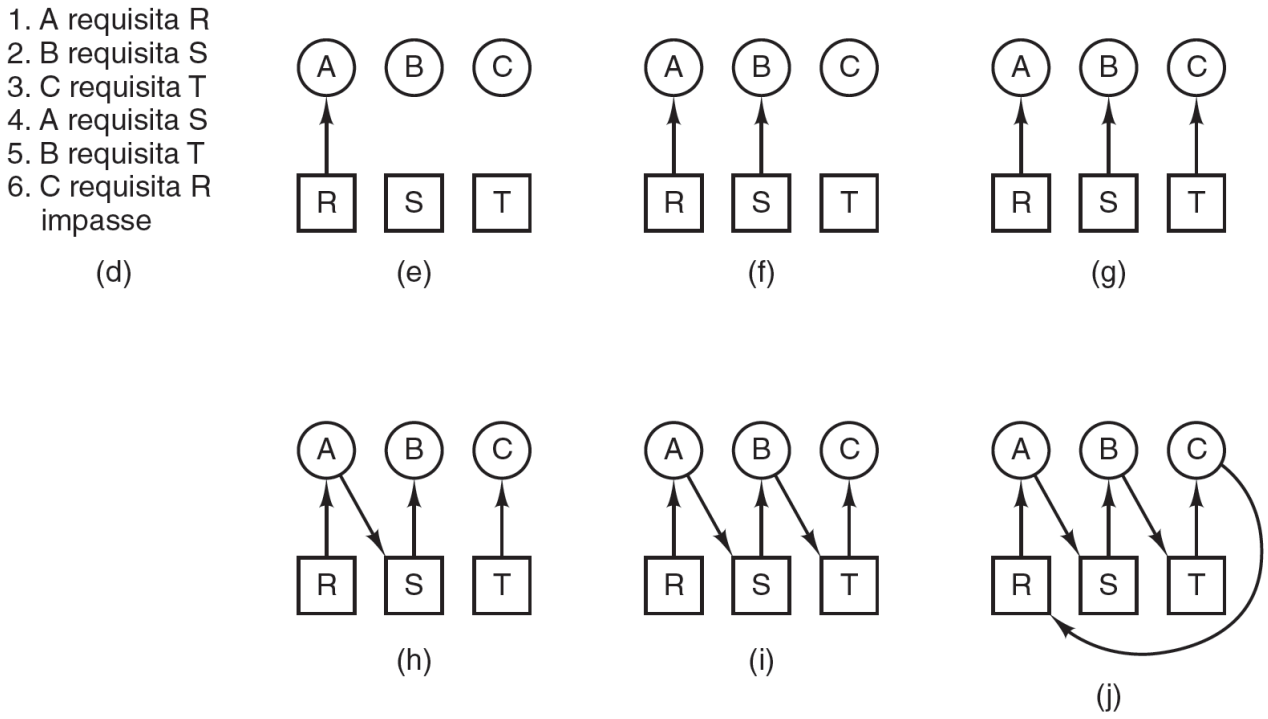


Figura 6.4 Exemplo de como um impasse ocorre e como pode ser evitado.

Modelagem de impasses

Mas o SO pode escalonar os processos como quiser!

Se o atendimento for capaz de gerar um impasse

⇒ SO pode suspender o processo, sem atender à requisição, até que a mesma possa ser atendida com segurança

Ex.: se o SO soubesse que o impasse era iminente, poderia suspender B e só deixar A e C executarem. (*Figs 6.4L a 6.4q*)

Modelagem de impasses

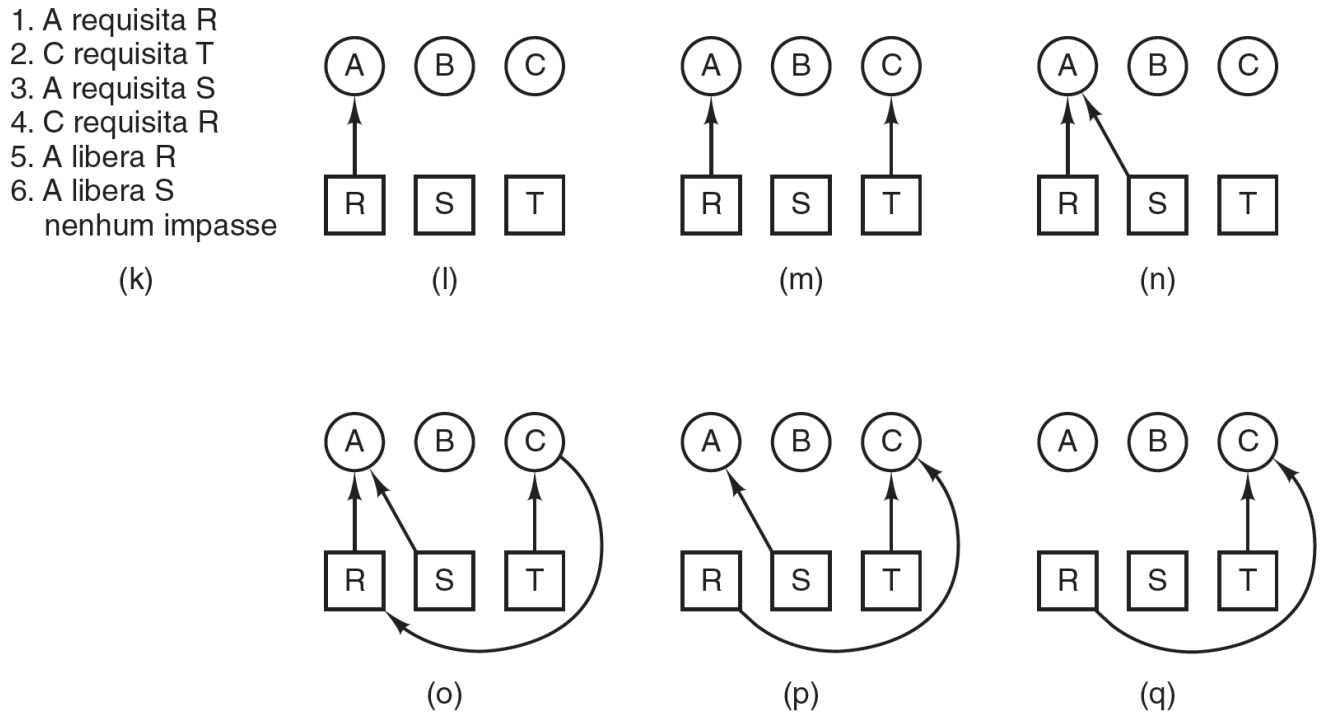


Figura 6.4 Exemplo de como um impasse ocorre e como pode ser evitado.

Modelagem de impasses

Conclusão

Grafo de recursos é uma poderosa ferramenta para verificar se uma sequência de requisição/liberação pode levar a um impasse

Estratégias para lidar com impasses

1. Ignorar o problema (não é boa ideia!).

2. Detecção e recuperação. Deixe os impasses ocorrerem, detecte-os e recupere-se deles.

3. Evitar, dinamicamente, que eles ocorram, por meio de alocação cuidadosa de recursos.

4. Prevenção, fazendo com que uma das quatro condições necessárias para o impasse não ocorram

Detecção e Recuperação de Impasses

Detecção de impasses com um recurso de cada tipo

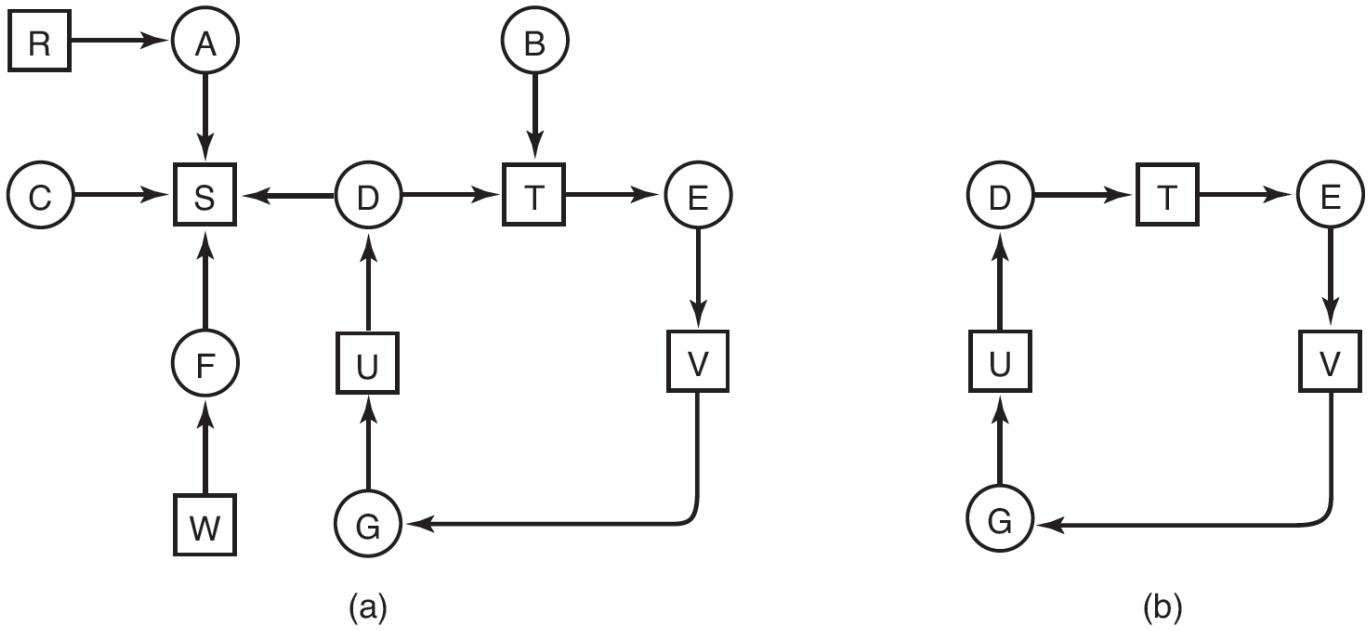


Figura 6.5 (a) Um gráfico de recursos. (b) Um ciclo extraído de (a).

Detecção e Recuperação de Impasses

Analisar situação de cada processo. O sistema está em impasse?

Se estiver, quais processos estão em impasse?

Procedimento:

1. Verificar se existem ciclos
2. Qualquer processo em um ciclo estará em impasse (na figura, D, E e G)

Obs: detecção visual é simples, mas para um sistema real deve-se ter um algoritmo formal

Detecção e Recuperação de Impasses

Algoritmo para detecção de impasse:

1. Para cada nó, N, no grafo, execute os cinco passos seguintes, usando N como nó inicial.

2. Inicialize uma lista vazia L e assinale todos os arcos como desmarcados.

3. Insira o nó atual no final de L, verifique se o nó aparece em L duas vezes. Se aparece, o gráfico contém um ciclo (listado em L) e o algoritmo termina.

4. A partir do nó dado, verifique se existe algum arco de saída desmarcado. Em caso afirmativo, vá para o passo 5; do contrário, vá para o passo 6.

Detecção e Recuperação de Impasses

Algoritmo para detecção de impasse (cont):

5. Escolha aleatoriamente um arco de saída desmarcado e marque-o. Então, siga esse arco para obter o novo nó atual e volte para passo 3.

6. Se esse nó for o inicial, o gráfico não conterá ciclo algum e o algoritmo terminará. Senão, o final foi alcançado. Remova-o e volte para o nó anterior, isto é, aquele que era atual antes desse. Marque-o como atual e volte ao passo 3.

Detecção e Recuperação de Impasses

Observações:

- ✓ Algoritmo faz uma busca em profundidade tomando cada nó como raiz do que se espera ser uma árvore
- ✓ quando torna a passar por um nó percorrido ⇒ ciclo
- ✓ se retornar ao nó raiz e não puder ir adiante

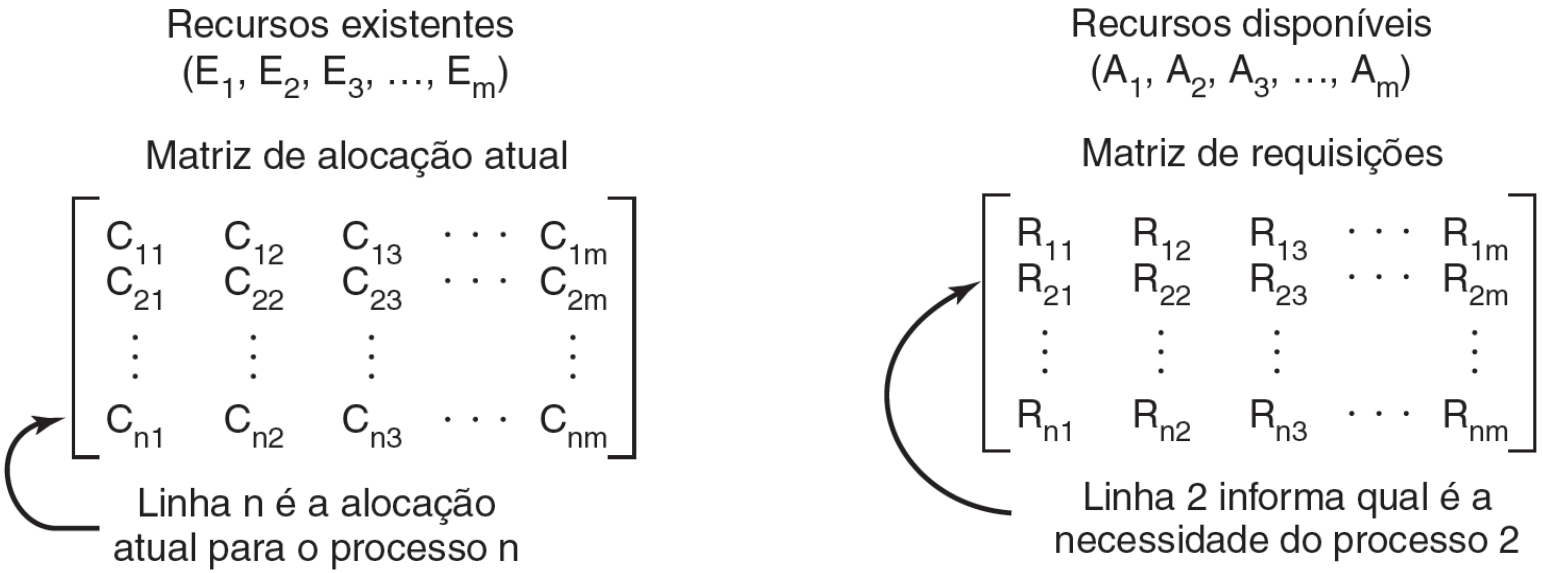
⇒ subgrafo alcançável a partir deste nó não tem ciclos.

Se essa propriedade for válida para todos os nós

⇒ **sistema não contém impasses**

Detecção e Recuperação de Impasses

Detecção de impasses com múltiplos recursos de cada tipo



Detecção e Recuperação de Impasses

Algoritmo de detecção de impasse:

1. Procure um processo desmarcado, P_i, para o qual a *i*-ésima linha de *R* seja menor ou igual a *A*.

2. Se esse processo for encontrado, diminua R_{ij} de A e adicione a *i*-ésima linha de C a A, marque o processo e volte para o passo 1.

3. Se não existir esse processo, o algoritmo terminará.

Detecção e Recuperação de Impasses

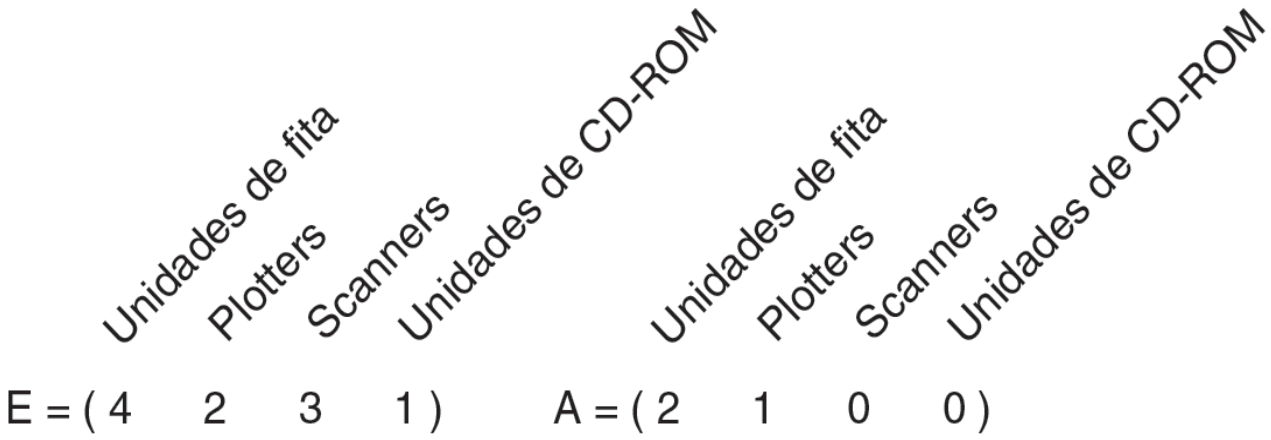


Figura 6.7 Um exemplo para o algoritmo de detecção de impasses.

Detecção e Recuperação de Impasses

Ex. 2:

Suponha agora que o processo 2 precise de 2 unidades de fita, uma unidade do plotter e uma unidade de CD-ROM, ou seja:

$R_2 = (2\ \ 1\ \ 0\ \ 1)$

$\Rightarrow$ IMPASSE!

Detecção e Recuperação de Impasses

Quando procurar pelos impasses?

Opções:

- Toda vez que acontecer uma requisição
  - ✓ Vantagem: detecção precoce
  - ✓ Desvantagem: caro em termos de tempo de CPU
- Procurar por eles a cada *k* minutos
- Procurar por eles quando a utilização da CPU cair (indica muitos impasses)

Recuperação de impasses

- Recuperação por preempção.
- Recuperação por retrocesso.
- Recuperação por eliminação de processos.

Recuperação de impasses

Recuperação por preempção:

Ex: SO utilizado em sistemas de processamento em lote

- ✓ Retira-se, provisoriamente e manualmente, um recurso de um processo e entrega-se a outro. Ex: uma impressora a laser
- ✓ Processo que perdeu o recurso temporariamente não deve ser afetado $\Rightarrow$ muito dependente do tipo de recurso
- ✓ A escolha do processo depende de qual deles tem recursos que podem ser facilmente devolvidos

  Ex.: impressora pode ser retomada. Mas, e um gravador de Blu-ray?

Recuperação de impasses

Recuperação por retrocesso:

- Processos geram pontos de salvaguarda **(*checkpoints*)** periodicamente, armazenando seu estado em um arquivo, para ser reiniciado posteriormente, se for necessário
- Estado: arquivo contém a imagem da memória e quais recursos estavam alocados ao processo naquele instante
- Cada ***checkpoint*** é escrito em um arquivo diferente para não sobrepor o anterior
- Quando um impasse é detectado, um dos processos que possui um recurso é retrocedido a um de seus ***checkpoints*** anteriores (trabalho feito a partir deste ponto é perdido)
- Se processo tentar adquirir o recurso novamente e este estiver ocupado, ele terá que esperar até o recurso estar disponível

Recuperação de impasses

Recuperação por eliminação de processos:

Obs: maneira mais grosseira (e mais simples) de eliminar um impasse

- Escolhe-se um processo do ciclo ou qualquer outro fora do ciclo que, ao ser eliminado, libere recursos suficientes para algum que esteja bloqueado
- Deve-se "matar" um processo que possa ser re-executado desde o início sem problemas. **Exemplo: compilador**

Contra-exemplo: processo que atualiza um banco de dados

soma "+1" a um registro. Se for executado novamente

$\Rightarrow$ **+1+1 = +2**

Evitando impasses

Na discussão anterior: processos requisitam todos os recursos

Na prática: recursos são requisitados um de cada vez

⇒ **Sistema deve decidir se fazer a alocação de um recurso é seguro ou não e somente alocar um recurso quando for seguro!**

Existe algum algoritmo que possa evitar os impasses fazendo sempre a escolha certa? (decidir se aloca ou não o recurso ao processo que o solicita)

Resposta: Sim, se algumas informações estiverem disponíveis antecipadamente!

Evitando impasses – trajetórias de recursos

Exemplo: processo A requisita a impressora no instante I_1 e a libera em I_3 e requisita o plotter no instante I_2 e o libera em I_4. Processo B requisita o plotter em I_5 e o libera em I_7 e requisita a impressora em I_6 e a libera em I_8

- Processador único: movimentos para norte ou para leste (nunca na diagonal, para sul ou para oeste)
- Regiões hachuradas: dois processos têm posse da impressora ou do plotter. **Impossível por causa da exclusão mútua**
- **Sistema deve evitar entrar no quadrado I_1-I_2/I_5-I_6, caso contrário entrará em um impasse na interseção de I_2 com I_6**
- Em ***t*** deve-se executar o processo A até que I_4 seja alcançado
- a partir daí, qualquer trajetória até **u** será segura

Evitando impasses

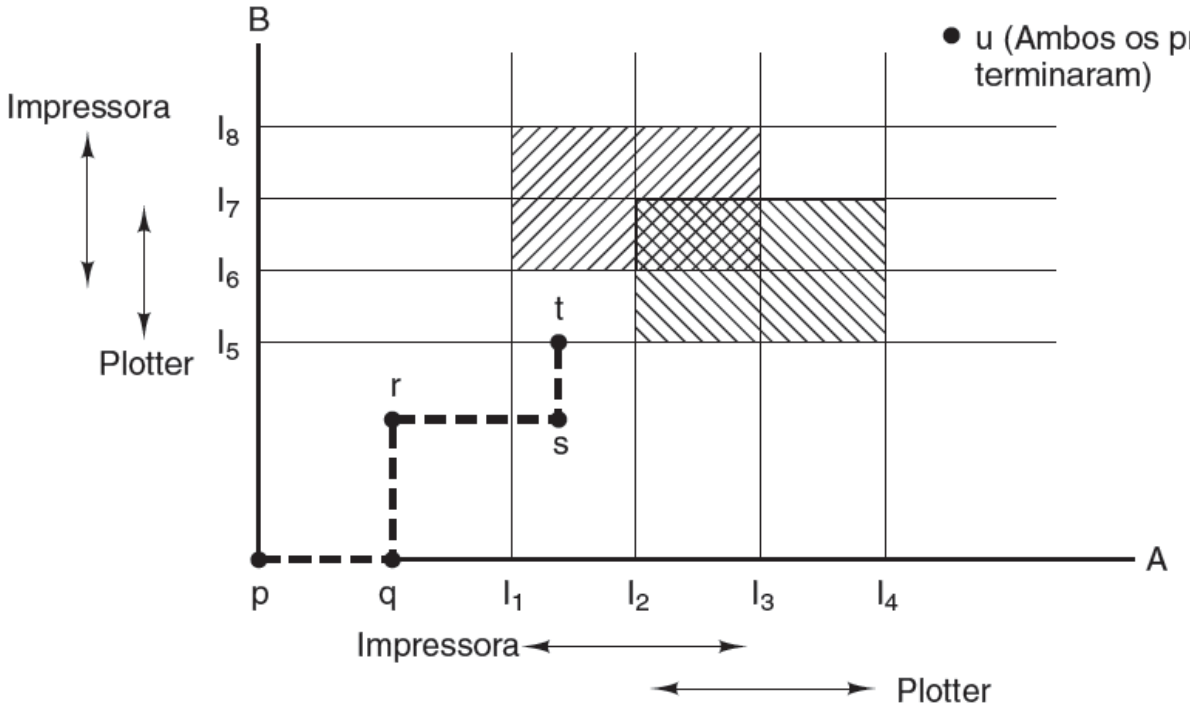


Figura 6.8 A trajetória de recursos de dois processos.

Estados seguros e inseguros

*"um estado é considerado **seguro** se ele não está em situação de impasse e se existe alguma ordem de escalonamento na qual podemos **garantir** que todo processo possa ser executado até sua conclusão, mesmo que, em algum momento, todos eles requisitem, de uma só vez, o máximo possível de recursos"*

Estados seguros e inseguros

Fig 6.9a - é um estado seguro pois existe uma sequência de execução/alocações de recursos que permite que todos os processos sejam concluídos

- B solicita dois recursos: fig. 6.9a → fig. 6.9b e executa até seu término

  ⇒ fig. 6.9b → fig. 6.9c

- Escalonador executa C

  ⇒ fig. 6.9d

- C é completado ⇒ **fig 6.9e**

⇒ A pode obter as 6 instâncias das quais ele precisa para completar sua execução!!!

Estados seguros e inseguros

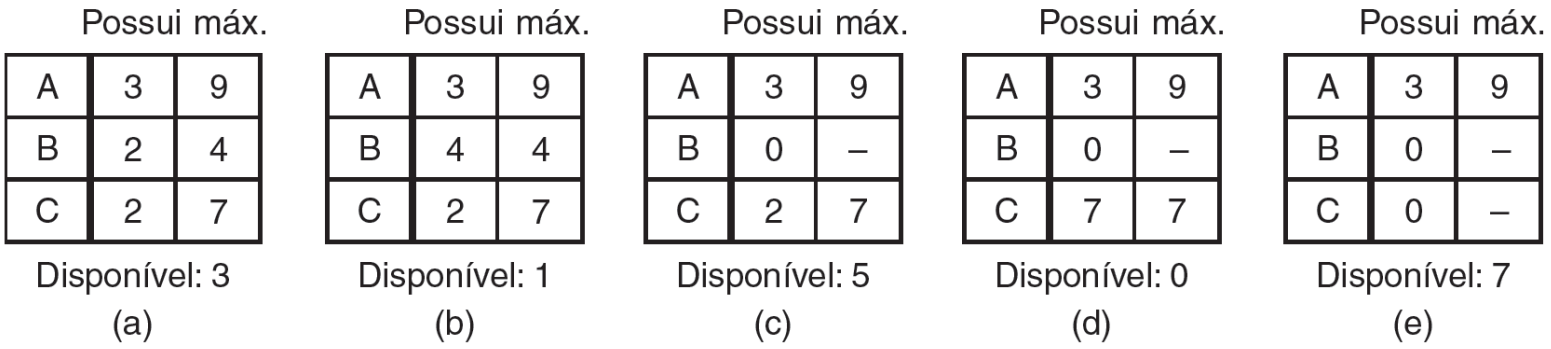


Figura 6.9 Demonstração de que o estado em (a) é seguro.

Estados seguros e inseguros

Outro exemplo: Fig 6.10a

- **A** requisita e obtém outro recurso ⇒ 6.10b
- Existe uma sequência de alocações que funcione?
- Escalonador executa **B** até este obter todos os recursos ⇒ 6.10c
- **B** é completado ⇒ 6.10d!!!! tem-se somente 4 instâncias disponíveis, mas cada um dos processos (**A e C**) necessitam de mais 5 instâncias

NÃO EXISTE SEQUÊNCIA QUE GARANTA A CONCLUSÃO!

⇒ quando o escalonador decidiu alocar um recurso ao processo A (6.10a para 6.10b), levou o sistema para um estado inseguro!!

Estados seguros e inseguros

	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)

Figura 6.10 Demonstração de que o estado em (b) é inseguro.

Estados seguros e inseguros

Observação:

Um estado inseguro não é uma situação de impasse

- A diferença para um **estado seguro** é que este pode ***garantir*** que todos os processos terminarão
- Na figura 6.10 é possível que o sistema execute sem atingir um impasse (por exemplo, o processo A pode liberar um recurso antes de pedir mais)
- Em um **estado inseguro** ***não se pode garantir*** que não haverá impasse

Algoritmo do banqueiro para um único recurso

- banqueiro (SO) verifica se a liberação do crédito (recurso) a algum cliente (processo) leva a algum estado inseguro
- em caso positivo, requisição é negada
- se levar a um estado seguro, é liberada

Algoritmo do banqueiro para um único recurso

Exemplo: figura 6.11

1º caso (a partir de 6.11b): liberação de 2 créditos para C

- C executa e libera os 4 créditos para o banqueiro
- A partir daí, banqueiro pode liberar 4 créditos para B ou D
- ….

2º caso (a partir de 6.11b): B solicita 1 crédito

⇒ situação de 6.11c (estado inseguro) –

– nenhum dos processos poderiam ser atendidos (executar até o final com a liberação de recursos novos)

⇒ solicitação negada!!

Algoritmo do banqueiro para um único recurso

	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)

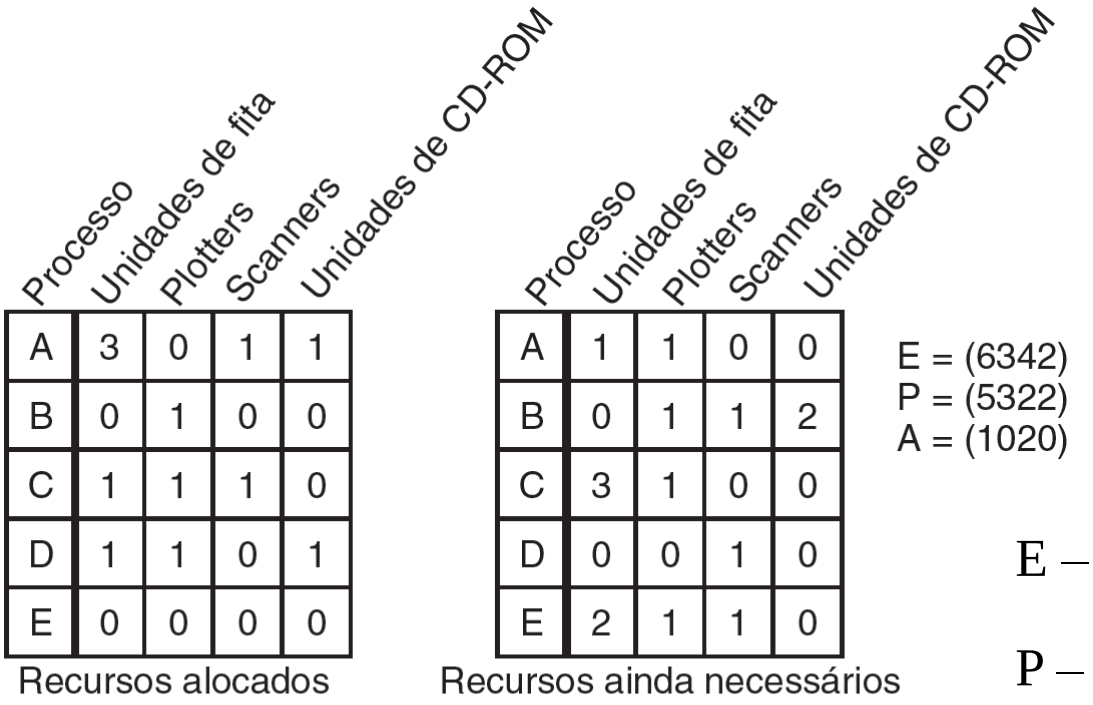
	Possui	máx.
A	1	6
B	2	5
C	2	4
D	4	7

Disponível: 1

(c)

Figura 6.11 Três estados de alocação de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

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)

E – recursos existentes

P – recursos alocados

A – recursos disponíveis

Figura 6.12 O algoritmo do banqueiro com múltiplos recursos.

Algoritmo do banqueiro para múltiplos recursos

O algoritmo que verifica se um estado é seguro:

1. Procure uma linha de R (requisições), cujas necessidades de recursos sejam $\leq$ A. Se não existir tal linha o sistema acabará por entrar em impasse, já que nenhum processo poderá ser executado por completo.

2. Considere que o processo da linha escolhida requer todos os recursos que necessita e termina. Marque o processo como terminado, acrescente todos os seus recursos ao vetor A.

3. Repita os passos 1 e 2 até que todos os processos estejam marcados como terminados (estado seguro) ou até que sobre um ou mais processos sem recursos disponíveis para executá-lo(s). Há um impasse, caso inseguro

Algoritmo do banqueiro para múltiplos recursos

No exemplo da figura 6.12:

– estado atual é seguro

– B solicita uma scanner (na 4ª edição, impressora!)

⇒ **Requisição pode ser atendida!!**

... pois o processo D pode terminar e devolver os recursos, habilitando A ou E a ser executado e assim sucessivamente

– após alocar o scanner a B, E tb solicita um scanner: o vetor A ficará (1 0 0 0), o que levaria a um impasse (nenhum processo poderá ser executado por completo)

⇒ **Requisição de E deve ser negada, por enquanto!**

Algoritmo do banqueiro para múltiplos recursos

Considerações / Problemas:

- ✓ processos raramente sabem, antecipadamente, o nº máximo de recursos que irão precisar
- ✓ nº de processos não é fixo. Depende do número de usuários no sistema
- ✓ recursos podem quebrar (qtde de recursos disponíveis pode variar)

Prevenção de impasses

Observação:

Evitar impasses dinamicamente é muito difícil pois requer informações sobre requisições futuras normalmente não disponíveis

O que fazer?

Tentar garantir que pelo menos uma das 4 condições necessárias para que um impasse aconteça nunca seja satisfeita

Prevenção de impasses

- Atacando a condição de exclusão mútua
- Atacando a condição de posse e espera
- Atacando a condição de não preempção
- Atacando a condição de espera circular.

Condições para impasse de recursos

1. **Condição de exclusão mútua** – Recurso está associado a um único processo ou está disponível.

2. **Condição de posse e espera** – Processos que detêm algum recurso podem requisitar novos recursos.

3. **Condição de não preempção** – Recursos fornecidos a um processo não podem ser tomados do mesmo. Têm que ser liberados pelo processo.

4. **Condição de espera circular** – Deve existir um encadeamento circular de dois ou mais processos; cada um à espera de outro recurso que está sendo usado pelo membro da cadeia.

Atacando a condição de exclusão mútua

Exemplo: dois processos que acessem uma impressora ao mesmo tempo:

Ideia: utilizar um ***spool de impressão***, onde somente o processo ***daemon*** atua, gerenciando o ***spool***. Como o ***daemon*** jamais solicita outros recursos, o impasse para a impressora poderia ser evitado.

Problema:

Daemons são programados para imprimir somente quando o arquivo completo está disponível. Neste caso, o espaço em disco reservado para o ***spool*** pode se esgotar, por ex, com dois processos cada um preenchendo metade do ***spool*** $\Rightarrow$ **impasse no espaço em disco.**

Solução: evitar alocar um recurso quando não for absolutamente necessário, para tentar minimizar o nº de processos que requisitam o recurso

Atacando a condição de posse e espera

Ideia: tentar impedir que processos que detenham algum recurso possam esperar por outros recursos

Processos alocariam todos os recursos dos quais precisam de uma só vez ou não alocariam

Problemas:

1. processos muitas vezes só conhecem os recursos dos quais vão precisar durante a execução
2. processo bloqueia recursos para uso futuro

⇒ recursos não serão usados de forma otimizada!

Atacando a condição de não preempção

Ideia: retomar os recursos alocados a um processo

Ex.: retomar uma impressora à força

⇒ **procedimento complexo!**

Solução:

armazenar a saída da impressora em um spool de impressão no disco ***(virtualização da impressora)***

Problema: criar impasse no disco…
mas, normalmente, espaço em disco não é problema.

Obs: alguns dispositivos não podem ser virtualizados.
Ex: registros em banco de dados ou tabelas do SO

Atacando a condição de espera circular

Solução 1:

Processo só deve ter acesso a um recurso de cada vez.
Caso necessite de outro, deve liberar o primeiro.

⇒ **Solução inaceitável para um processo que quer copiar um arquivo muito grande de uma fita para a impressora, por exemplo**

Solução 2:

Numerar os recursos. Processos só podem solicitar os recursos em ordem númerica crescente

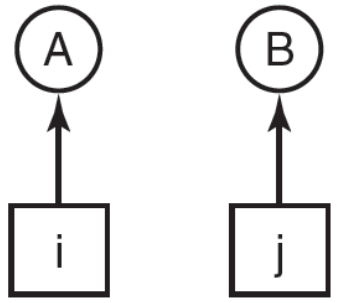
⇒ **impede ciclos**

Atacando a condição de espera circular

Exemplo:

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

(a)



(b)

Figura 6.13 (a) Recursos ordenados numericamente. (b) Um gráfico de recursos.

Existe impasse quando processo A requisita recurso j e processo B requisita recurso i!

Se i > j A não tem permissão de requisitar o recurso j e se j > i B não tem permissão para requisitar o recurso i

Atacando a condição de espera circular

- ✓ Também funciona com mais de dois processos: processo acabará ou irá requisitar recursos de ordem maiores
- ✓ A cada instante, um dos recursos será o de ordem mais alta e algum processo está de posse dele
- ✓ Quando o processo termina, libera o recurso de mais alta ordem

⇒ Todos os processos irão finalizar

Problema:

às vezes é difícil encontrar uma ordem numérica para uma grande quantidade de recursos mais abstratos (registros de banco de dados, por ex.) de forma a satisfazer todos os processos.

Abordagens para prevenir impasses

Condição	Abordagem contra impasses
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

Tabela 6.1 Resumo das abordagens para prevenir impasses.

Outras questões

- Travamento em duas fases
- Impasses de comunicação
- Livelock
- Condição de inanição

Travamento em duas fases

É uma abordagem para tentar impedir impasse

Exemplo de problema: em um sistema de banco de dados, muitos processos podem querer bloquear registros para posterior atualização ⇒ **pode ocorrer impasse!**

Primeira fase: processo tenta bloquear todos os registros dos quais precisa, um de cada vez

Segunda fase: se for bem sucedido na primeira fase, realiza as atualizações dos registros. Se não for, libera todos os registros e reinicia a primeira fase.

Obs: **a reiniciação é inadmissível para sistemas de tempo real ou em um protocolo de comunicação, se o sistema já leu mensagens da rede**

Impasse de comunicação

A envia solicitação a **B** e bloqueia até que **B** envie resposta. Resposta se perde! **A** fica bloqueado! **B** fica esperando resposta de **A**!

$\Rightarrow$ **Impasse de comunicação**

Não existem recursos envolvidos!

$\Rightarrow$ Impasse não pode ser evitado pela ordenação de recursos e nem por escalonamento cuidadoso

Solução: usar um temporizador (timeout)!

Impasse de recursos em sistemas de comunicação

A espera **B** ter lugar no buffer, que espera **C**, que espera **D**, que espera **A**!

⇒ ***impasse de recursos!***

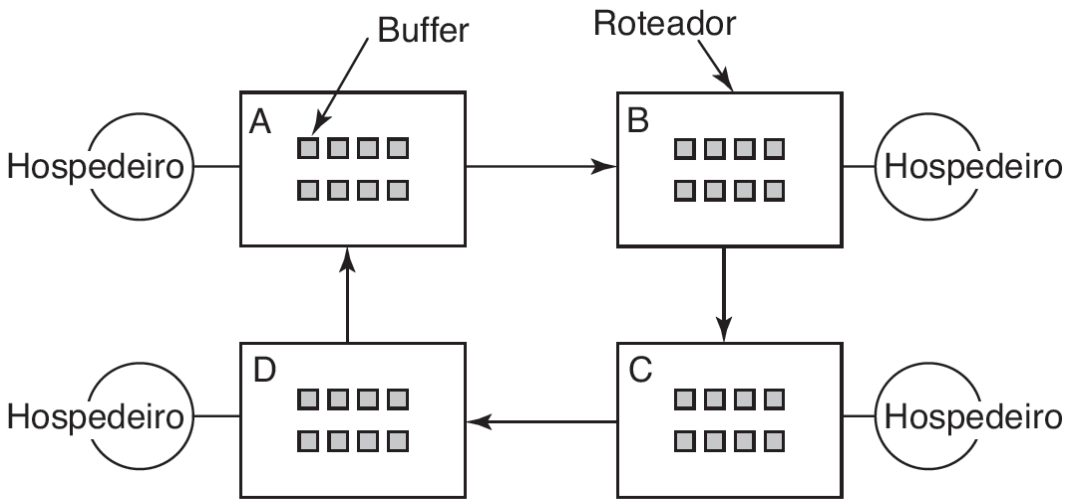


Figura 6.14 Um impasse de recurso em uma rede.

Livelock

Enter_region:
primitiva de polling! Se tentativa de obter o recurso falha, processo tenta novamente.

Processo A obtém recurso 1 e processo B obtém recurso 2. Processos não bloqueiam (não há impasse), mas ficam indefinidamente tentando obter o recurso necessário para terminar sua execução!

```
void process_A(void) {
        enter_region(&resource_1);
        enter_region(&resource_2);
        use_both_resources( );
        leave_region(&resource_2);
        leave_region(&resource_1);
}

void process_B(void) {
        enter_region(&resource_2);
        enter_region(&resource_1);
        use_both_resources( );
        leave_region(&resource_1);
        leave_region(&resource_2);
}
```

Figura 6.15 A espera ocupada que pode acarretar um livelock.

Condição de Inanição

Algumas políticas para prevenir impasses e decidir quem recebe um determinado recurso, e quando, podem prejudicar algum processo, que nunca é atendido em sua requisição, mesmo que não exista impasse

Ex: centenas de processos solicitando a impressora

- Algoritmo usado: menor arquivo é impresso
- A cada momento podem chegar mais processos com pequenos arquivos para serem impressos

$\Rightarrow$ **processo com um arquivo grande nunca terá a oportunidade de imprimi-lo:** ***condição de inanição***

Solução para este problema: usar FCFS!