

Deadlock

Um problema de concorrência

Parte dos slides: Sistemas Operacionais
Modernos – 2ª Edição, Pearson Education

Interação entre Processos

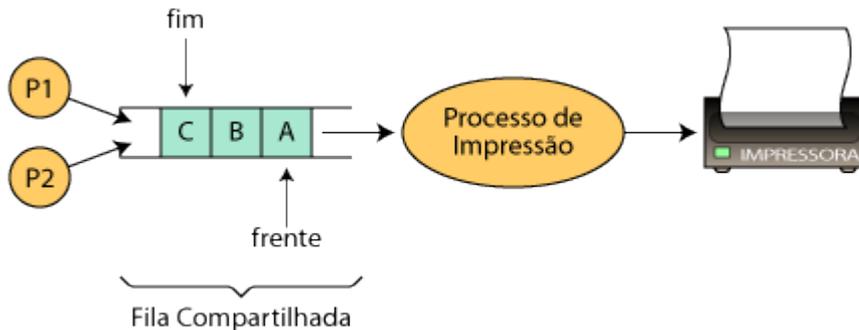
- Processos que executam em paralelo podem trocar informações entre si
- Níveis de interação (concorrência):
 - **Processos Independentes:** Não existe troca de informações
 - **Processos Cooperativos (Comunicativos):** Um processo produz informações utilizadas por outros processos
 - Exemplo: *Shell Pipes* (ls || more)
 - **Processos Competitivos:** Vários processos competem pelo uso do mesmo recurso
 - Exemplo: Um tocador de CD e outro de MP3 rodando em paralelo

Comunicação entre Processos

Condições de Corrida

Exemplo: Fila de impressão

- Qualquer processo que queira imprimir precisa colocar o seu documento na fila de impressão (compartilhada)
- O processo de impressão retira os documentos na ordem em que chegaram na fila enviando-os à impressora



- Se a fila é compartilhada, isto significa que seus dados, assim como os indicadores de frente e fim da fila também o são

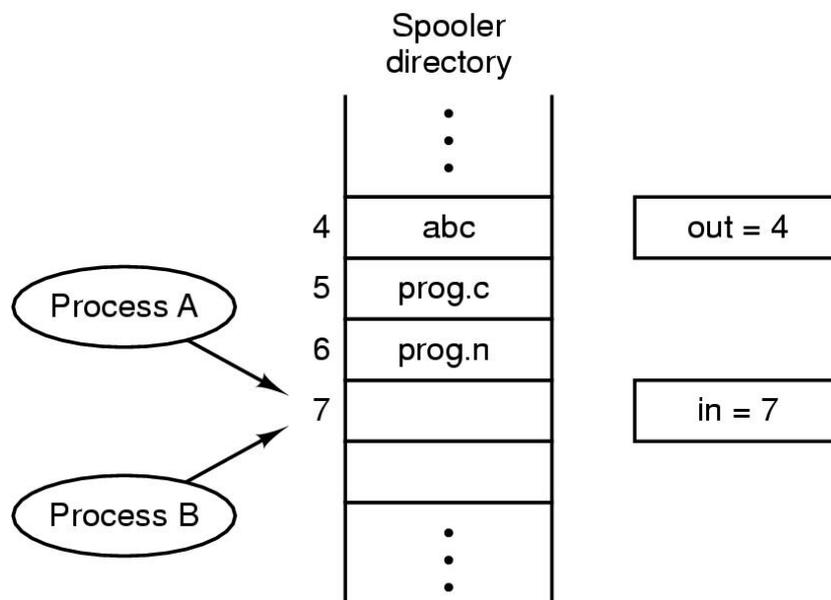
- Um processo que queira imprimir um documento precisa executar a operação de inserção na fila, que envolve basicamente os seguintes comandos:
 - fim++ (incrementa o indicador do fim da fila)
 - coloca documento na posição do novo fim da fila
- Se dois processos resolvem simultaneamente imprimir um documento e se, por causa do compartilhamento do tempo do processador, o primeiro processo for interrompido (por ter acabado o seu tempo - *quantum*) entre os comandos 1 e 2, então o segundo processo não poderá imprimir seu arquivo (tentou inserir seu documento na fila antes que o primeiro processo tivesse acabado de o fazer), e a fila ficará em um estado inválido
- Lembrando: esta situação se chama "condição de corrida"

<http://www.ppgia.pucpr.br/~laplima/aulas/so/materia/processos.html>

Comunicação entre Processos

Condições de Corrida

2 processos querendo acessar memória compartilhada ao mesmo tempo.
Quem "chega" primeiro?



- Para evitar as condições de corrida, precisamos implementar algoritmos de **exclusão mútua** de execução e, para tanto, definimos as **regiões críticas** do processo

Problemas com Concorrência

- Não-determinismo
- Dependência de Velocidade
- *Starvation*
- *Deadlock*

Recursos (1)

- Exemplos de recursos:
 - Impressoras
 - 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 (2)

- *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 (3)

- Sequência 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 ou associado a um processo ou 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* (1)

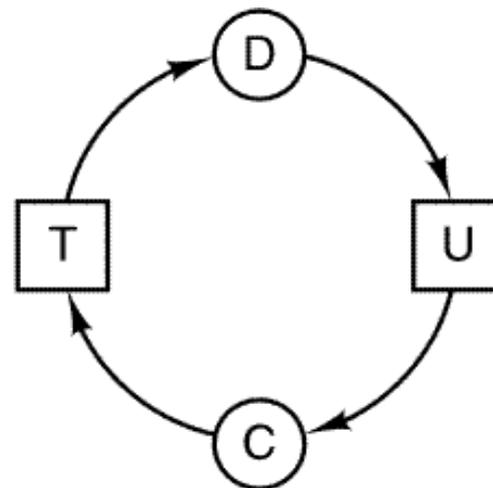
- Modelado com grafos dirigidos



(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

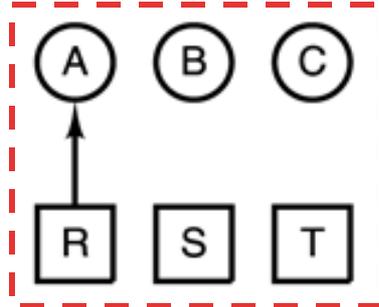
Modelagem de *Deadlock* (2)

Estratégias para tratar *Deadlocks*

1. ignorar por completo o problema
2. detecção e recuperação
3. **evitação dinâmica**
 - alocação cuidadosa de recursos
4. prevenção
 - negação de uma das quatro condições necessárias

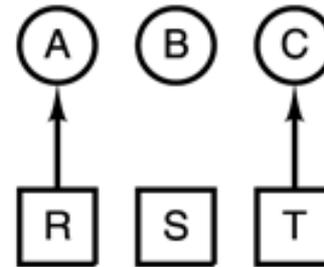
Modelagem de *Deadlock* (3)

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

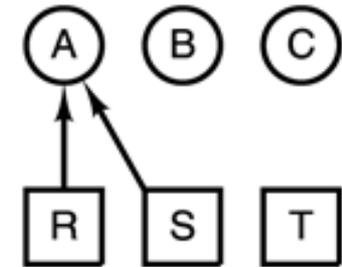


(k)

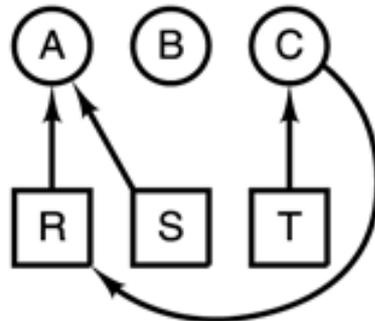
(l)



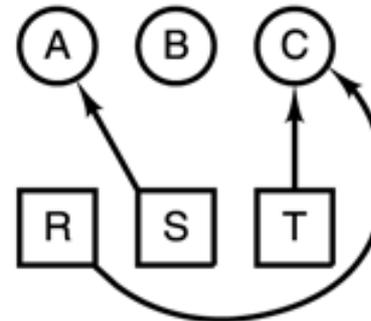
(m)



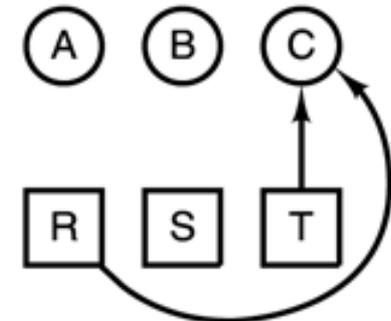
(n)



(o)



(p)



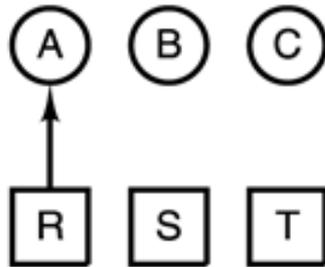
(q)

Como pode ser **evitado** um *deadlock*

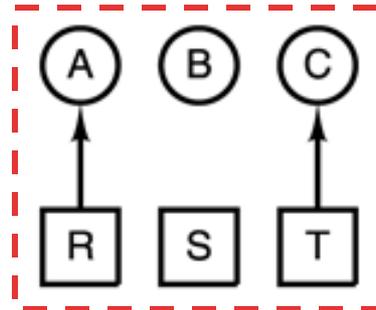
Modelagem de *Deadlock* (3)

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

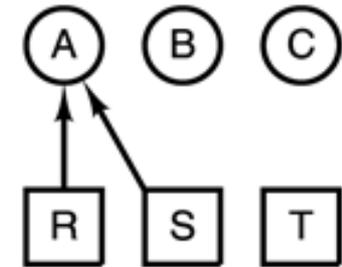
(k)



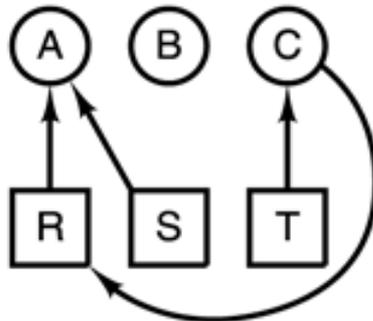
(l)



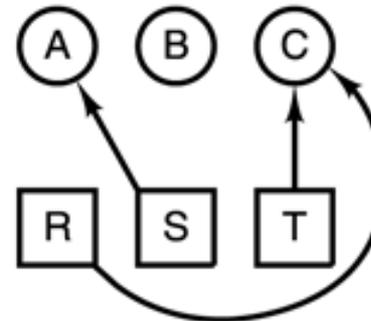
(m)



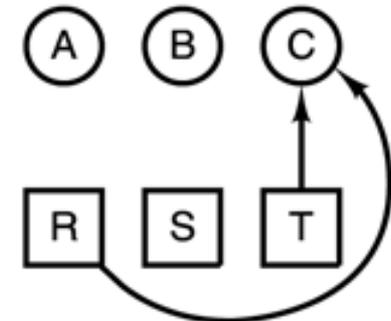
(n)



(o)



(p)



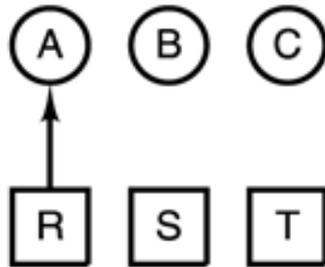
(q)

Como pode ser **evitado** um *deadlock*

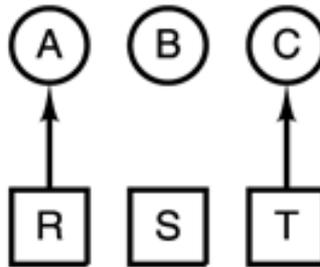
Modelagem de *Deadlock* (3)

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

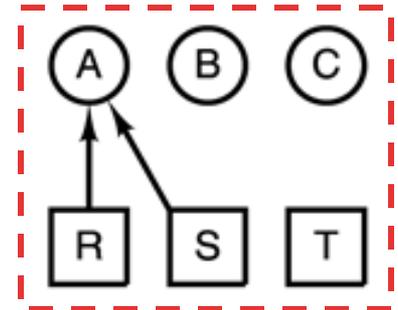
(k)



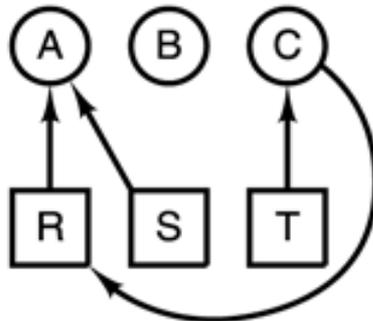
(l)



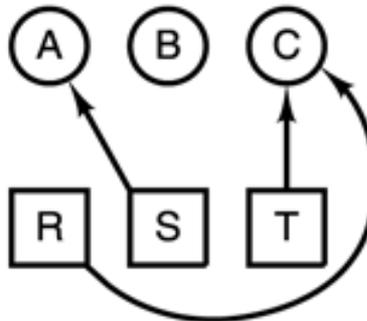
(m)



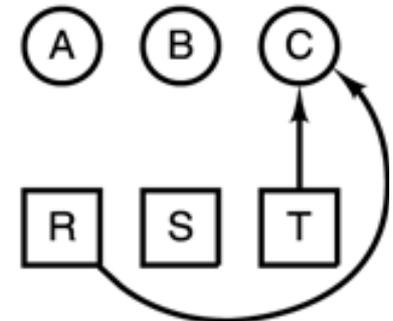
(n)



(o)



(p)

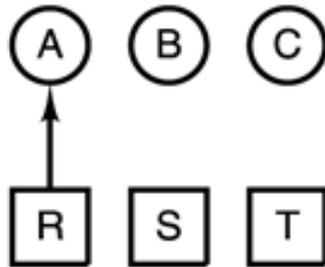


(q)

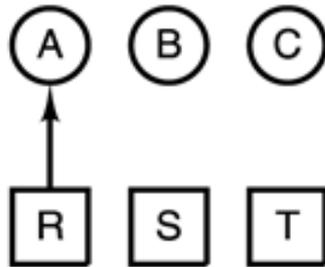
Como pode ser **evitado** um *deadlock*

Modelagem de *Deadlock* (3)

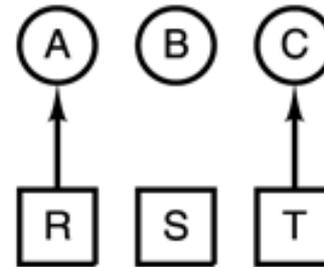
1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock



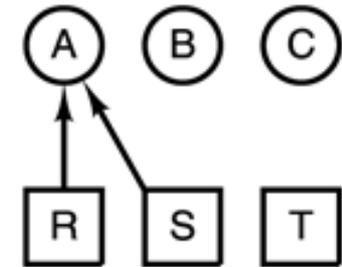
(k)



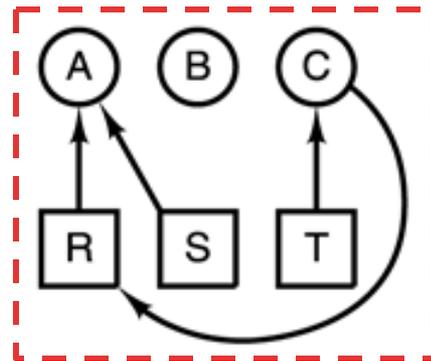
(l)



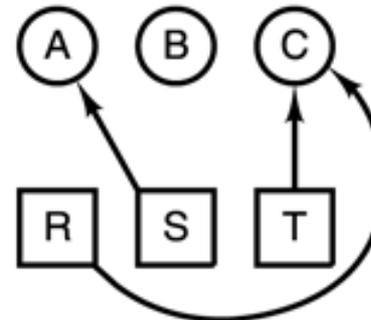
(m)



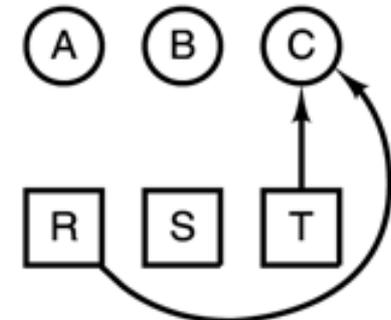
(n)



(o)



(p)

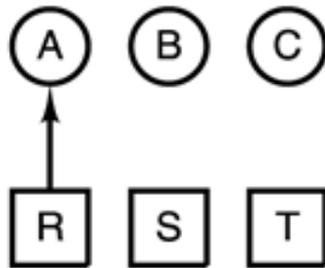


(q)

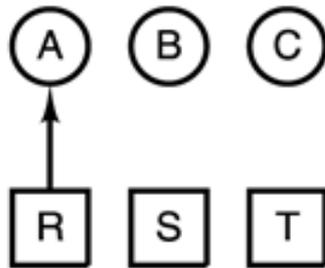
Como pode ser **evitado** um *deadlock*

Modelagem de *Deadlock* (3)

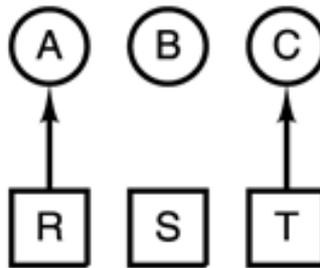
1. A requisita R
 2. C requisita T
 3. A requisita S
 4. C requisita R
 5. A libera R
 6. A libera S
- nenhum deadlock



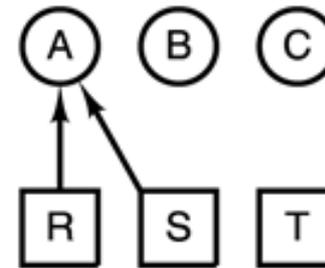
(k)



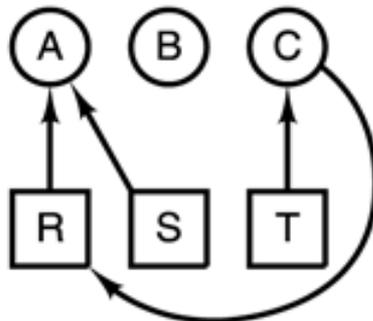
(l)



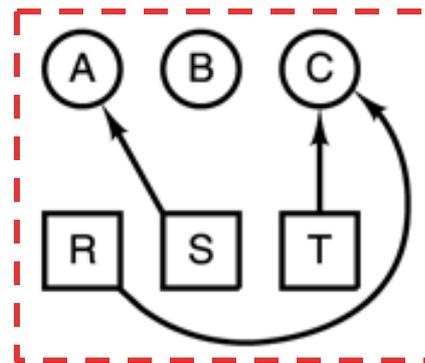
(m)



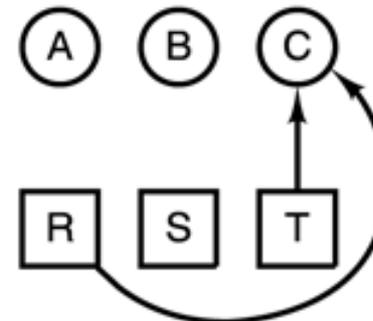
(n)



(o)



(p)



(q)

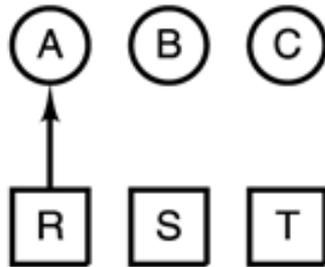
Como pode ser **evitado** um *deadlock*

Modelagem de *Deadlock* (3)

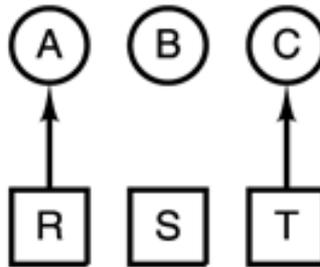
1. A requisita R
2. C requisita T
3. A requisita S
4. C requisita R
5. A libera R
6. A libera S

nenhum deadlock

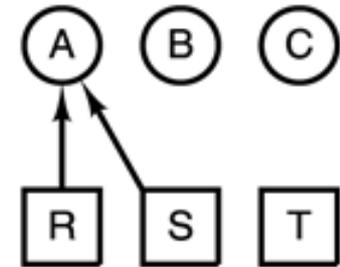
(k)



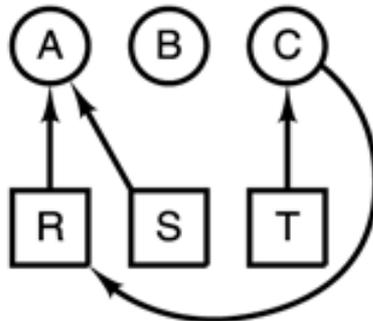
(l)



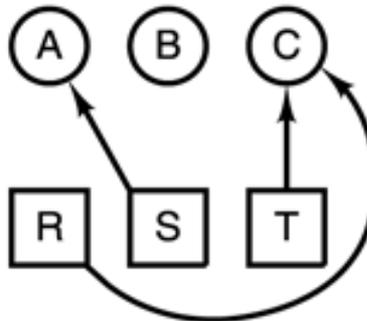
(m)



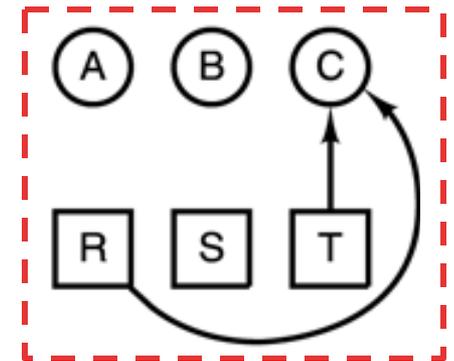
(n)



(o)



(p)



(q)

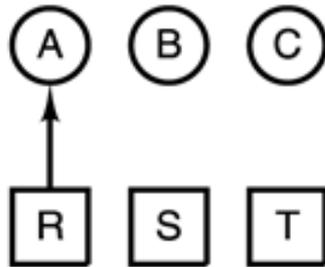
Como pode ser **evitado** um *deadlock*

Modelagem de *Deadlock* (3)

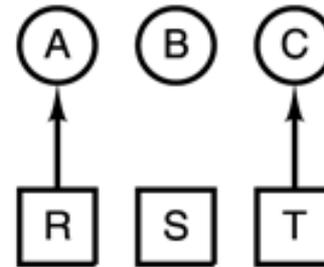
1. A requisita R
2. C requisita T
3. A requisita S
4. C requisita R
5. A libera R
6. A libera S

nenhum deadlock

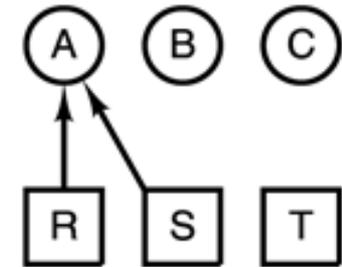
(k)



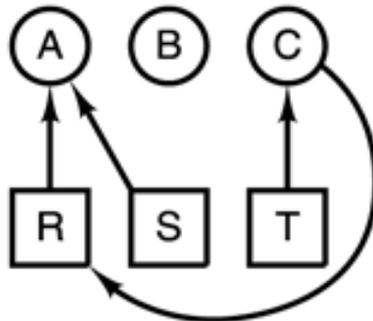
(l)



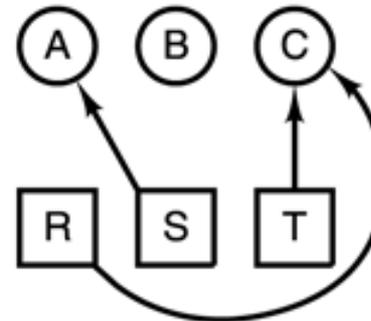
(m)



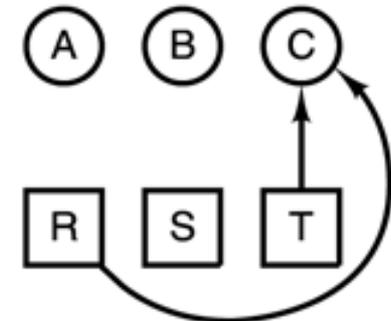
(n)



(o)



(p)



(q)

Como pode ser **evitado** um *deadlock*

Estratégias para tratar *Deadlocks*

1. ignorar por completo o problema
2. detecção e recuperação
3. evitação dinâmica
 - alocação cuidadosa de recursos
4. prevenção
 - negação de uma das quatro condições necessárias

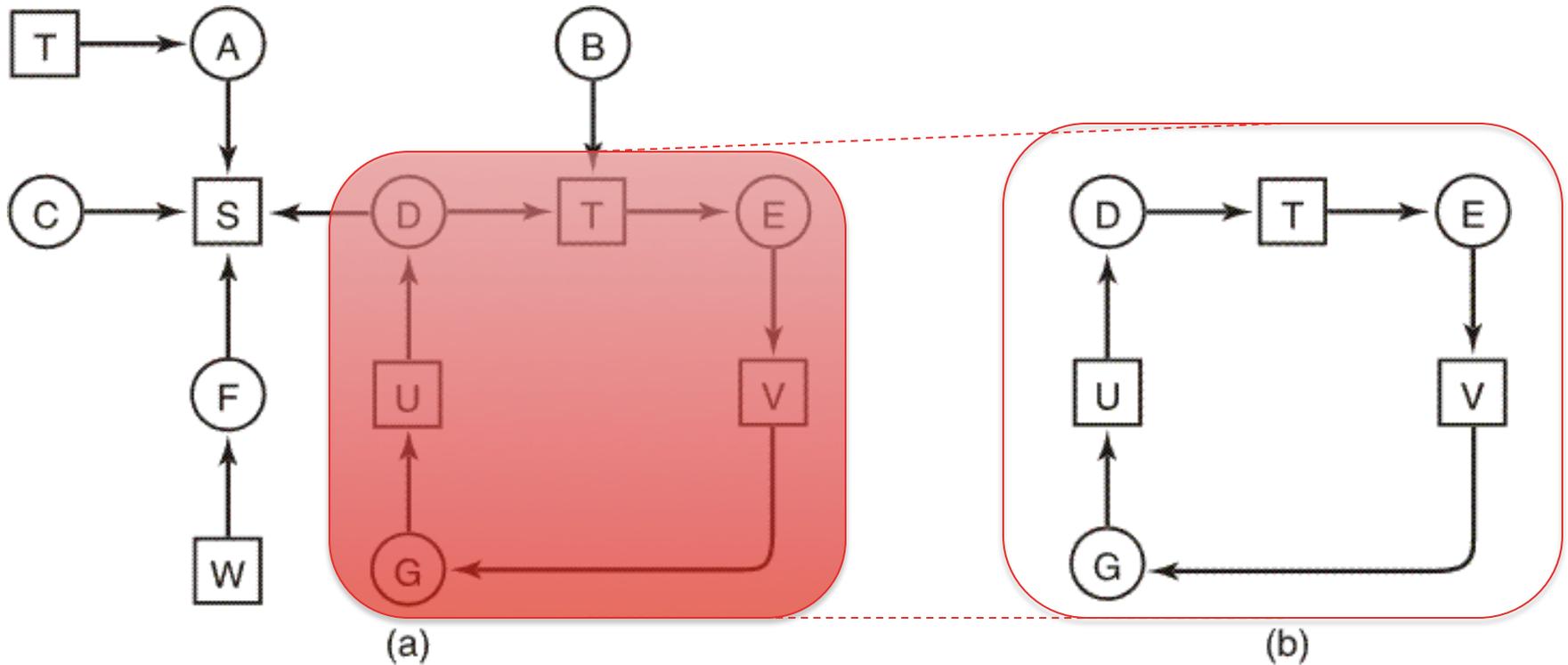
“Algoritmo” do Avestruz

- 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

Estratégias para tratar *Deadlocks*

1. ignorar por completo o problema
2. **detecção e recuperação**
3. evitação dinâmica
 - alocação cuidadosa de recursos
4. **prevenção**
 - negação de uma das quatro condições necessárias

Detecção com um Recurso de Cada Tipo



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

Recuperação de Deadlock

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

Estratégias para tratar *Deadlocks*

1. ignorar por completo o problema
2. detecção e recuperação
3. evitação dinâmica
 - alocação cuidadosa de recursos
4. **prevenção**
 - negação de uma das quatro condições necessárias



Quatro Condições para *Deadlock*

1. **Condição de exclusão mútua**
 - todo recurso está ou associado a um processo ou 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

Pearson Education Sistemas Operacionais Modernos – 2ª Edição 6

Prevenção de Deadlock

Atacando a Condição de Exclusão Mútua

- Alguns dispositivos (como uma impressora) podem fazer uso de *spool* [“fila”]
 - 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
- Variação:
 - processo deve desistir de todos os recursos
 - para então requisitar todos os que são **imediatamente** necessários

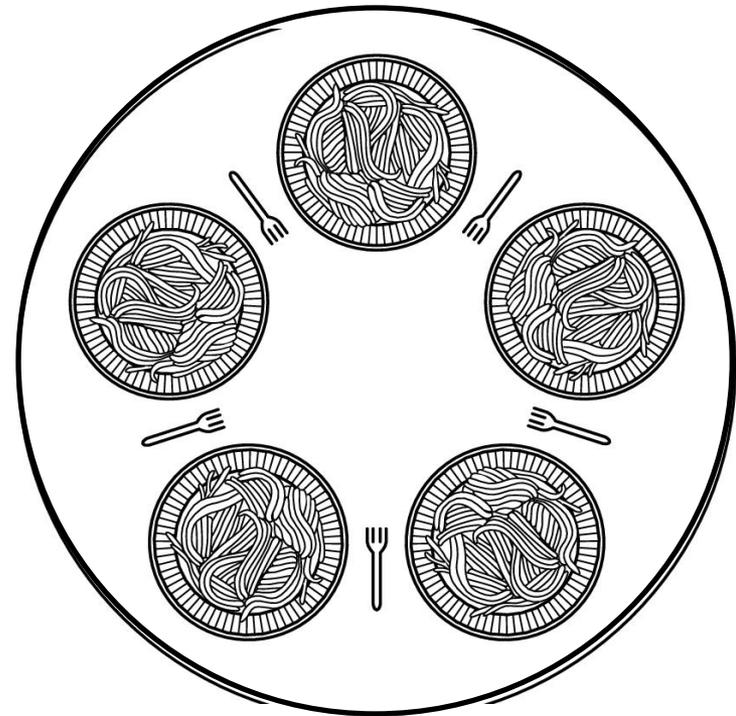
Prevenção de Deadlock

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

O clássico problema dos 'Filósofos Jantantes'

- Filósofos comem/ pensam
- Comem espaguete e precisam de 2 garfos
- Pegam um garfo de cada vez
- Como prevenir *deadlock* ?



```

#define N          5          /* number of philosophers */
#define LEFT      (i+N-1)%N  /* number of i's left neighbor */
#define RIGHT     (i+1)%N   /* number of i's right neighbor */
#define THINKING  0          /* philosopher is thinking */
#define HUNGRY    1          /* philosopher is trying to get forks */
#define EATING    2          /* philosopher is eating */
typedef int semaphore;      /* semaphores are a special kind of int */
int state[N];              /* array to keep track of everyone's state */
semaphore mutex = 1;       /* mutual exclusion for critical regions */
semaphore s[N];           /* one semaphore per philosopher */

void philosopher(int i)    /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {        /* repeat forever */
        think();          /* philosopher is thinking */
        take_forks(i);    /* acquire two forks or block */
        eat();            /* yum-yum, spaghetti */
        put_forks(i);     /* put both forks back on table */
    }
}

```

Solução (parte 1)

```

void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)                                    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i)                                         /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

```

/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */

```

```

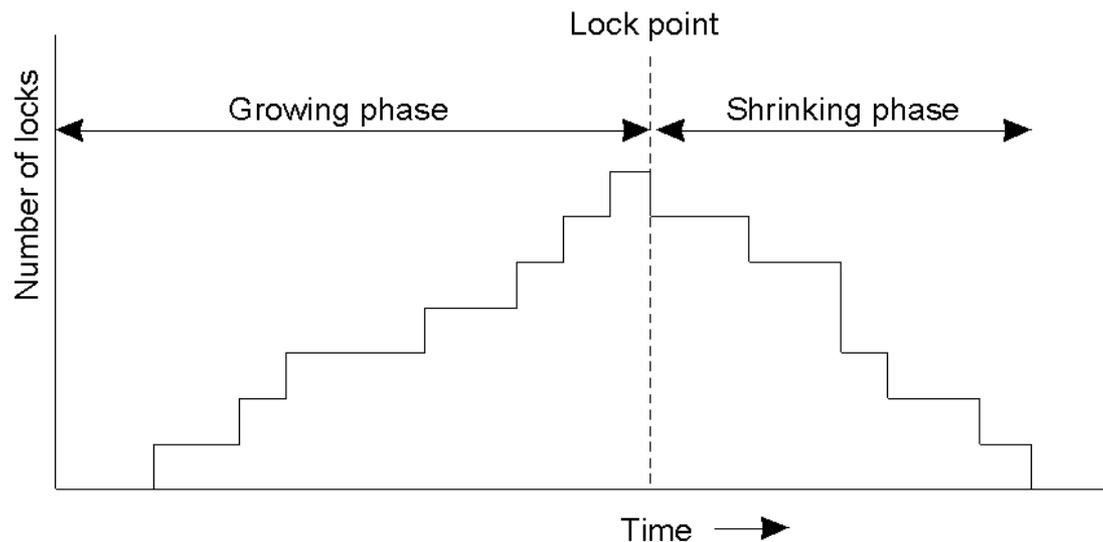
/* i: philosopher number, from 0 to N-1 */
/* enter critical region */
/* philosopher has finished eating */
/* see if left neighbor can now eat */
/* see if right neighbor can now eat */
/* exit critical region */

```

Solução (parte 2)

Two-Phase Locking

- Cada **transação** tem uma fase de requisição de *locks* (*growing phase*)
- Uma segunda fase libera *locks* (*shrinking phase*)



Distributed Systems: Principles and Paradigms
© Tanenbaum and van Steen 2002

O Modelo de Transação

```
BEGIN_TRANSACTION  
reserve REC -> RIO;  
reserve RIO -> POA;  
reserve POA -> BSB;  
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION  
reserve REC -> RIO;  
reserve RIO -> POA;  
reserve POA -> BSB full =>  
ABORT_TRANSACTION
```

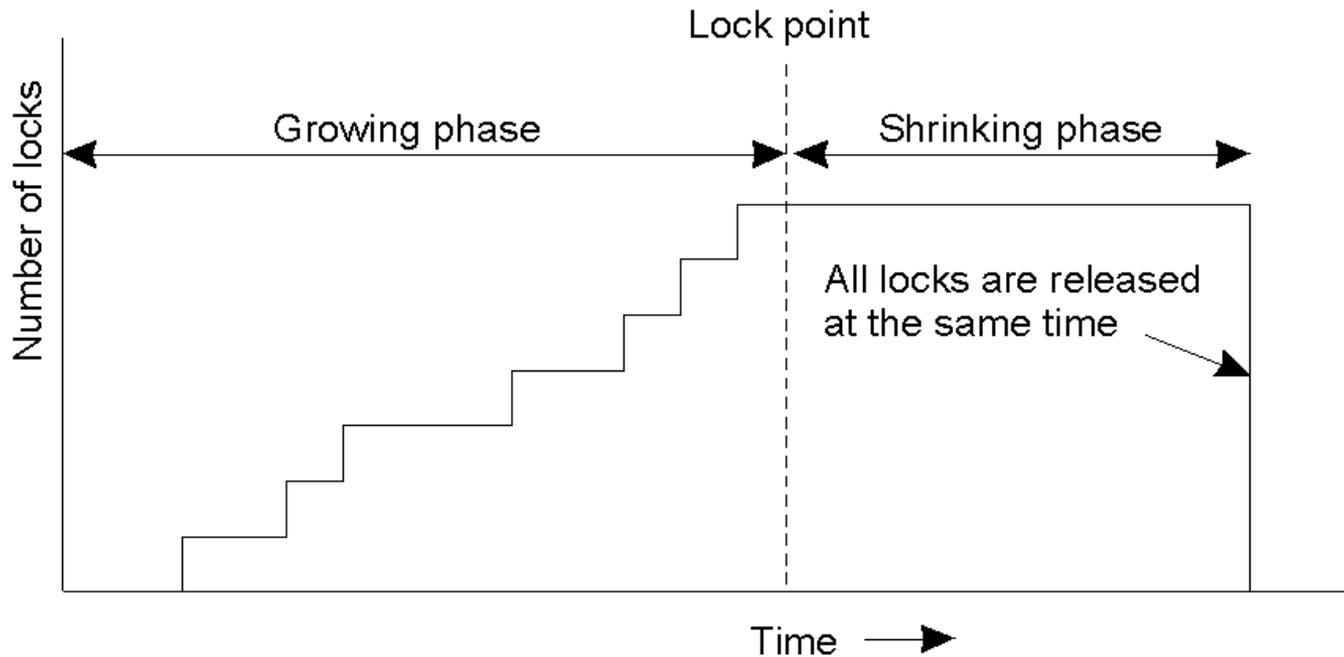
(b)

- a) Transação para a reserva de 3 vôos *commits*
- b) Transação aborta (*rollback*) quando o terceiro vôo não está disponível

Two-Phase Locking (2)

- Transações podem **abortar**
 - Então é necessário execução estrita para prevenir leituras erradas ou escritas prematuras
 - Assim, quaisquer *locks* conseguidos durante uma transação são sustentados até que a transação termine ou aborte ➡ ***strict two-phase locking***

Strict Two-Phase Locking



Distributed Systems: Principles and Paradigms
© Tanenbaum and van Steen 2002

Abordagens para Controle de Concorrência

1. A maioria dos sistemas usam esquemas de **locking**
 2. Ordenação baseada em **timestamps**
- 1 e 2 são abordagens **pessimistas**
 - Em **two-phase locking** e **ordenação baseada em timestamps**, o servidor detecta conflitos entre transações a cada acesso a item de dado
 - **Ordenação baseada em timestamps** é melhor para transações de leitura
 - **Two-phase locking** é melhor quando as operações são predominantemente de atualizações (escrita)
 - Esquemas **otimistas** permitem que uma transação prossiga até que termine, para então o servidor fazer uma checagem para descobrir se houve conflito
 - se sim, a transação terá que ser refeita