ACH2025

Laboratório de Bases de Dados

Aula 17

Controle de Concorrência

Professora:

- **Fátima L. S. Nunes**

USP

SISTEMAS DE INFORMAÇÃO

EACH

Conceitos - relembrando

- ✓ Transação: unidade de execução do programa que acessa/atualiza vários itens de dados.
  - Em geral, iniciada por um programa do usuário, escrita em linguagem de alto nível.
  - Delimitada pelas instruções: ***begin transaction*** e ***end transaction***
  - Para garantir integridade dos dados, SGBD deve manter as propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade.

Conceitos - relembrando

- ✓ Propriedades ACID:
  - **Atomicidade**: todas as operações executadas ou nenhuma delas.
  - **Consistência**: execução de transação isolada (sem outra transação simultânea) deve manter consistência dos dados.
  - **Isolamento**: uma transação não "percebe" outra transação – para uma determinada transação, ou transação terminou antes dela ou começou depois.
  - **Durabilidade**: após uma transação completada, mudanças persistem no BD, mesmo se houver falhas no sistema.

Conceitos - relembrando

- ✓ Transação
- ✓ Para garantir integridade dos dados, SGBD deve manter as propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade.
- ✓ Isolamento???

Conceitos

- ✓ Transação
- ✓ Para garantir integridade dos dados, SGBD deve manter as propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade.
- ✓ Isolamento: uma transação não “percebe” outra transação – para uma determinada transação, ou transação terminou antes dela ou começou depois.
- ✓ Esquemas de controle de concorrência: mecanismos que controlam **interação entre transações simultâneas**.
- ✓ Baseados na propriedade de seriação.
- ✓ Seriação?

Conceitos

- ✓ Transação
- ✓ Para garantir integridade dos dados, SGBD deve manter as propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade.
- ✓ Isolamento: uma transação não “percebe” outra transação – para uma determinada transação, ou transação terminou antes dela ou começou depois.
- ✓ Esquemas de controle de concorrência: mecanismos que controlam interação entre transações simultâneas.
- ✓ Baseados na propriedade de seriação.
- ✓ Seriação: *schedules* podem ser transformados em seriais por meio da troca da ordem de operações consecutivas.

Protocolos baseados em bloqueios

- ✓ Para garantir seriação:
  - itens de dados acessados de maneira mutuamente exclusiva ➔ enquanto uma transação acessa um dado, outra não pode modificá-lo.
  - método mais comum: bloqueio
- ✓ Bloqueios:
  - compartilhado (chamado de **S**): se transação T_i tem um bloqueio ***S*** sobre um item *Q*, T_i pode ler, mas não pode escrever Q.
  - exclusivo (chamado de **X**): se transação T_i tem um bloqueio ***X*** sobre um item *Q*, T_i pode ler e escrever Q.

Protocolos baseados em bloqueios

- ✓ Transação deve solicitar ao **gerenciador de controle de concorrência** o bloqueio no modo apropriado, dependendo das operações que deve realizar.
- ✓ Transação só pode prosseguir quando tem bloqueio concedido.
- ✓ Função de compatibilidade – denominada comp (A,B) sobre conjunto de modos de bloqueio:
  - considerando dois modos de bloqueio A e B e duas transações diferentes T_i e T_j:
    - T_j mantém bloqueio do modo B sobre um dado Q.
    - Se T_i puder receber um bloqueio no modo A, dizemos que o modo A é compatível com o modo B.

	S (compartilhado)	**X (exclusivo)**
S (compartilhado)	verdadeiro	falso
X (exclusivo)	falso	falso

Protocolos baseados em bloqueios

- ✓ instruções para transação:
  - **lock-s(Q):** solicitar bloqueio compartilhado sobre o dado Q
  - **lock-x(Q):** solicitar bloqueio exclusivo sobre o dado Q
  - **unlock(Q)**: desbloquear item de dado Q

- ✓ Se um dado estiver bloqueado em um modo incompatível, transação tem que esperar.

- ✓ Uma transação pode desbloquear um dado que tinha bloqueado anteriormente, mas nem sempre é bom que desbloqueie imediatamente ➔ necessário garantir seriação.

Execuções simultâneas

T1

```
lock-x(B);
read(B);
B := B – 50;
write (B);
unlock(B);
lock-x(A);
read(A)
A := A + 50;
write(A);
unlock(A);
```

T2

```
lock-s(A);
read(A)
unlock(A);
lock-s(B);
read(B);
unlock(B);
display(A+B);
```

Supondo valor inicial de A e B = 100 e 200, respectivamente, qual o valor final de A+B se transações forem executadas em série?

Execuções simultâneas

T1

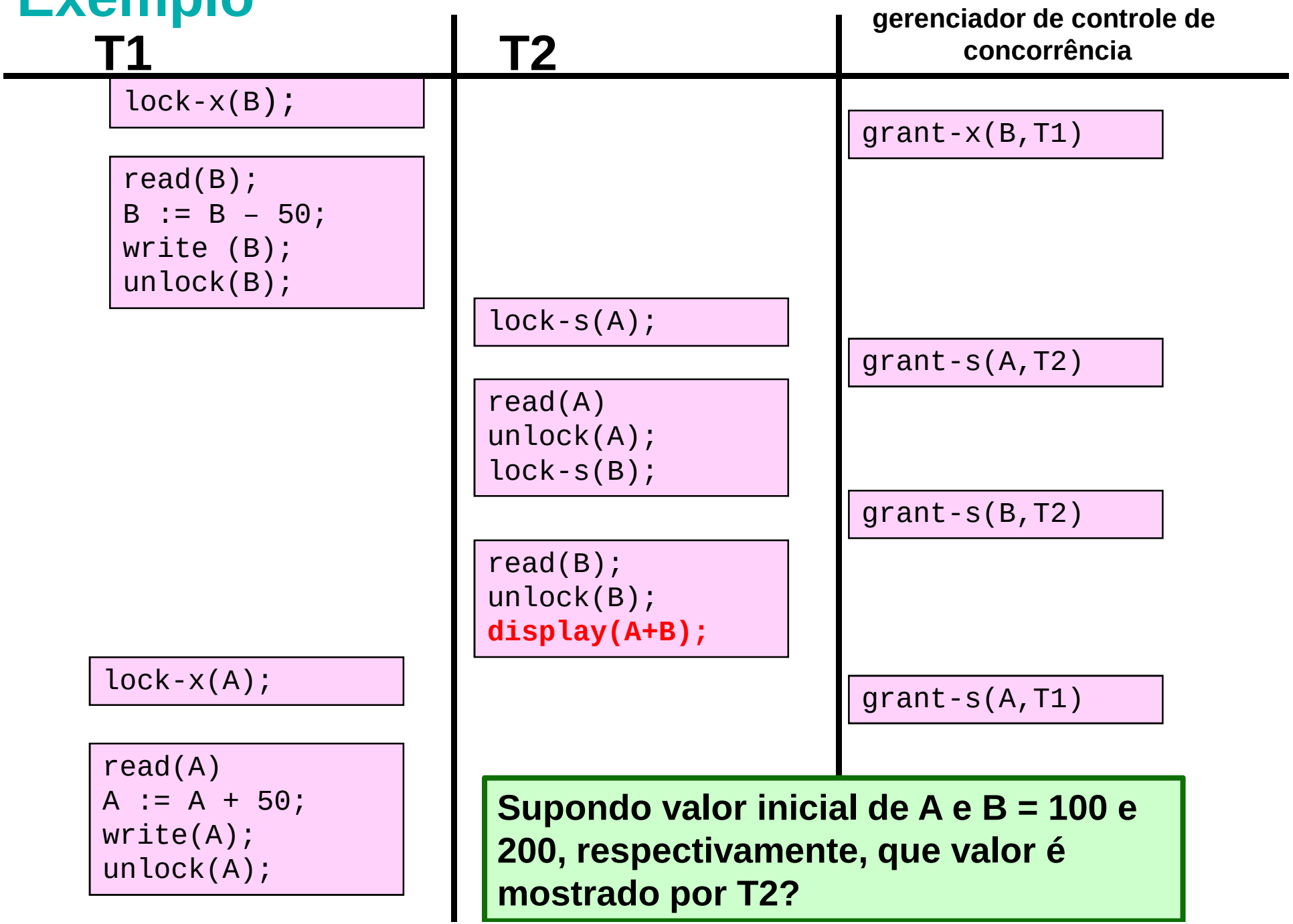
```
lock-x(B);
read(B);
B := B – 50;
write (B);
unlock(B);
lock-x(A);
read(A)
A := A + 50;
write(A);
unlock(A);
```

T2

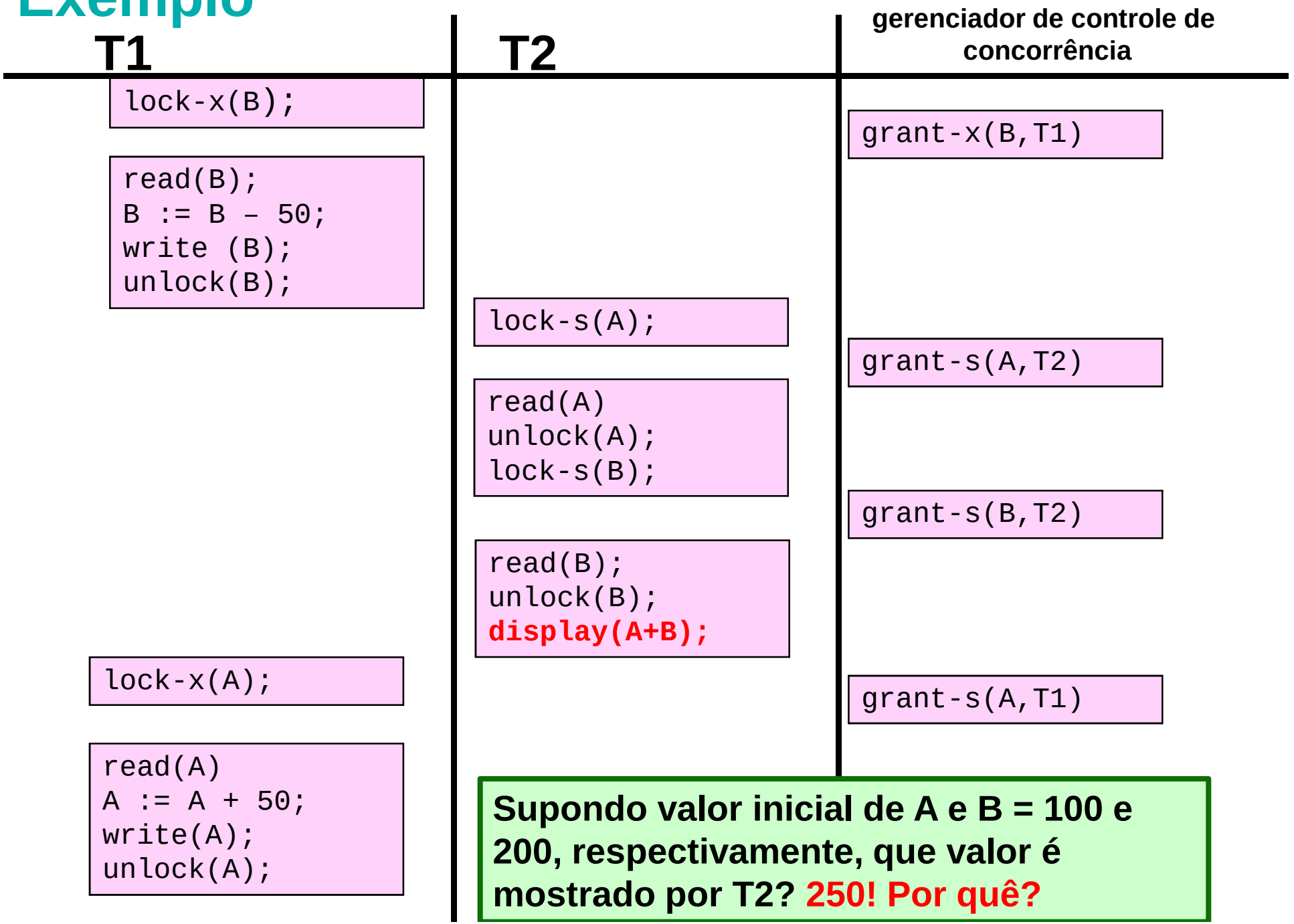
```
lock-s(A);
read(A)
unlock(A);
lock-s(B);
read(B);
unlock(B);
display(A+B);
```

Supondo valor inicial de A e B = 100 e 200, respectivamente, qual o valor final de A+B se transações forem executadas em série? 300 (soma de A + B)

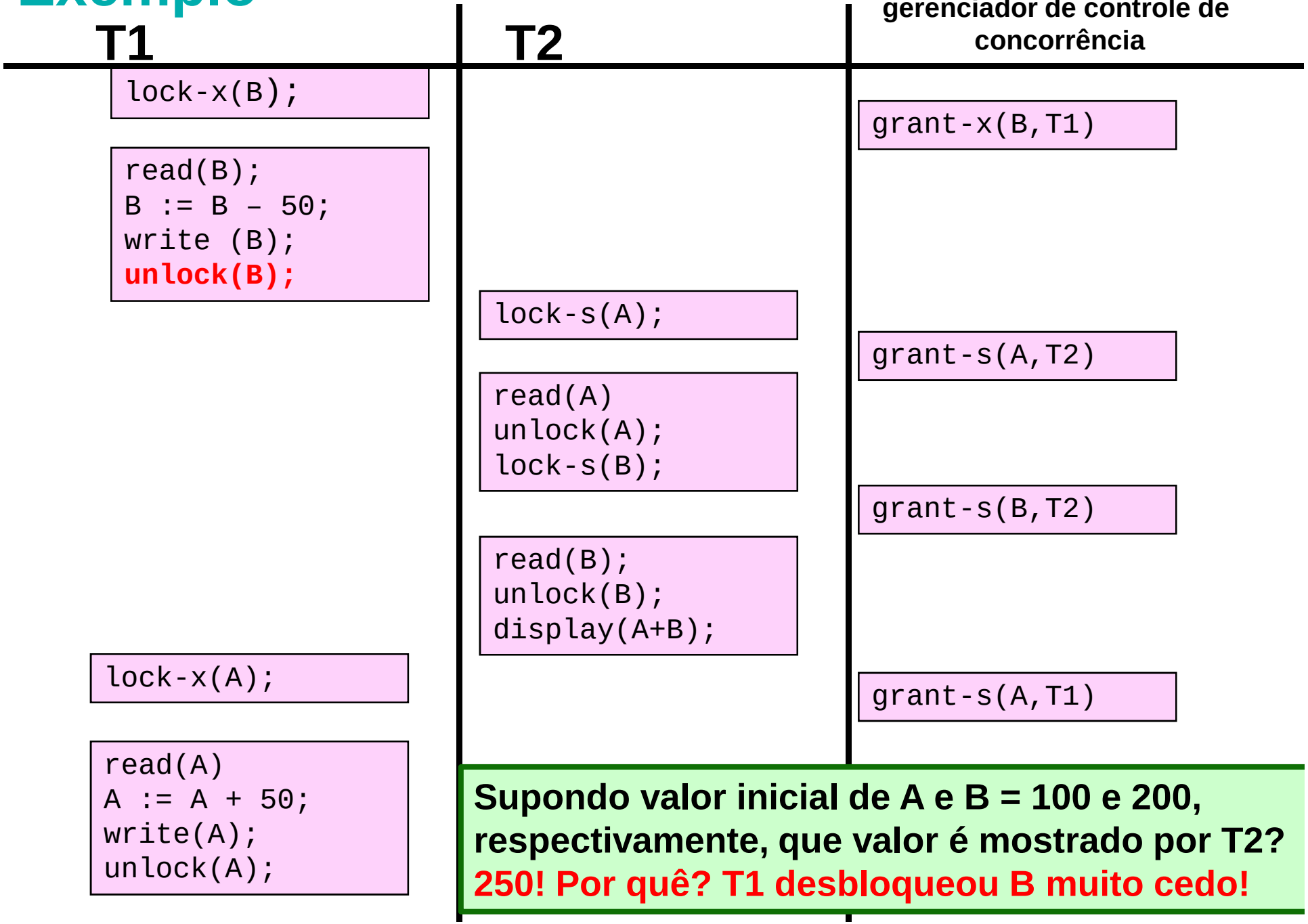
Exemplo


T1
T2
gerenciador de controle de concorrência
lock-x(B);
grant-x(B,T1)
read(B);
B := B – 50;
write (B);
unlock(B);
lock-s(A);
grant-s(A,T2)
read(A)
unlock(A);
lock-s(B);
grant-s(B,T2)
read(B);
unlock(B);
display(A+B);
lock-x(A);
grant-s(A,T1)
read(A)
A := A + 50;
write(A);
unlock(A);
Supondo valor inicial de A e B = 100 e 200, respectivamente, que valor é mostrado por T2?

Exemplo


T1
T2
gerenciador de controle de concorrência
lock-x(B);
grant-x(B,T1)
read(B);
B := B – 50;
write (B);
unlock(B);
lock-s(A);
grant-s(A,T2)
read(A)
unlock(A);
lock-s(B);
grant-s(B,T2)
read(B);
unlock(B);
display(A+B);
lock-x(A);
grant-s(A,T1)
read(A)
A := A + 50;
write(A);
unlock(A);
Supondo valor inicial de A e B = 100 e 200, respectivamente, que valor é mostrado por T2? 250! Por quê?

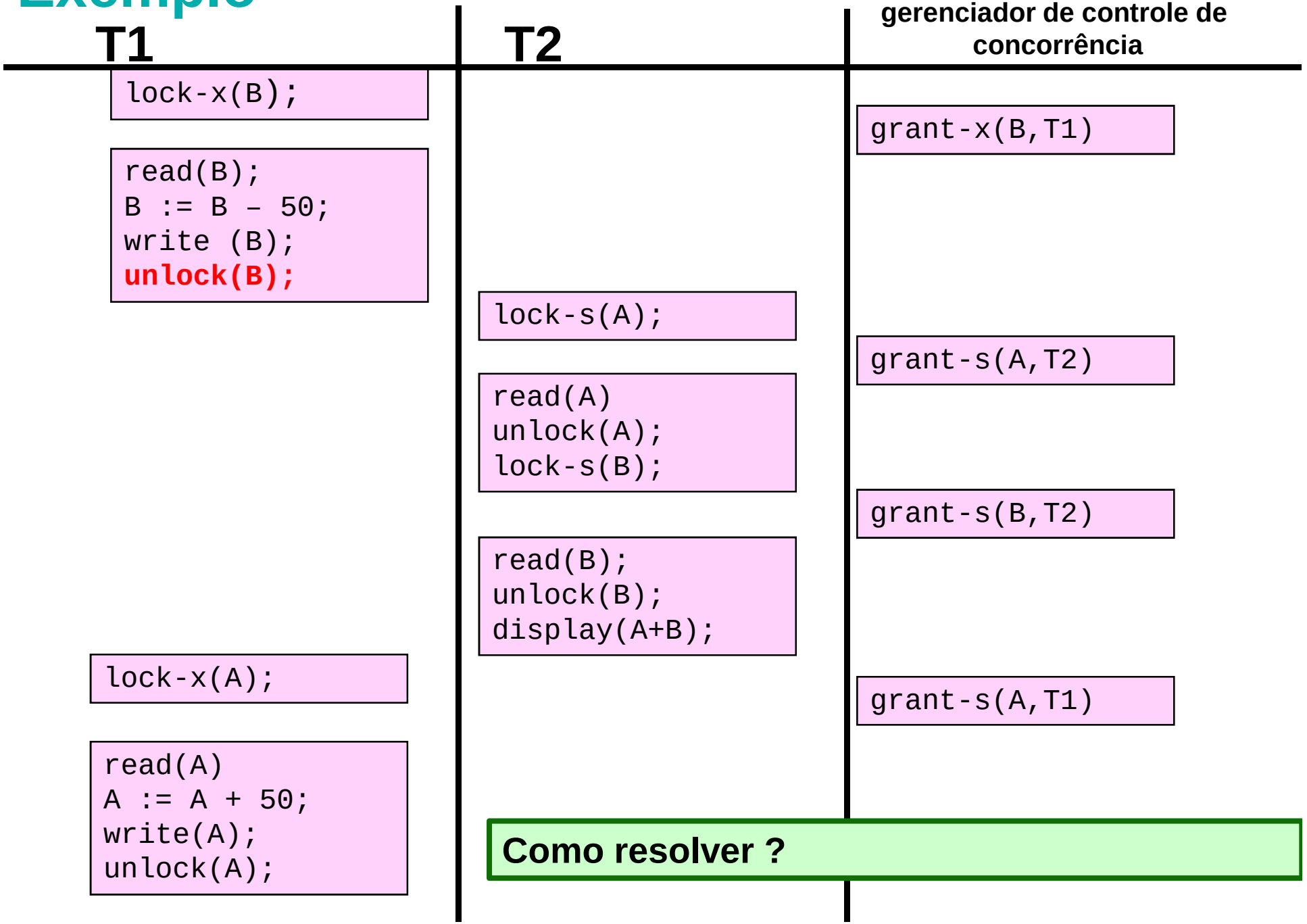
Exemplo



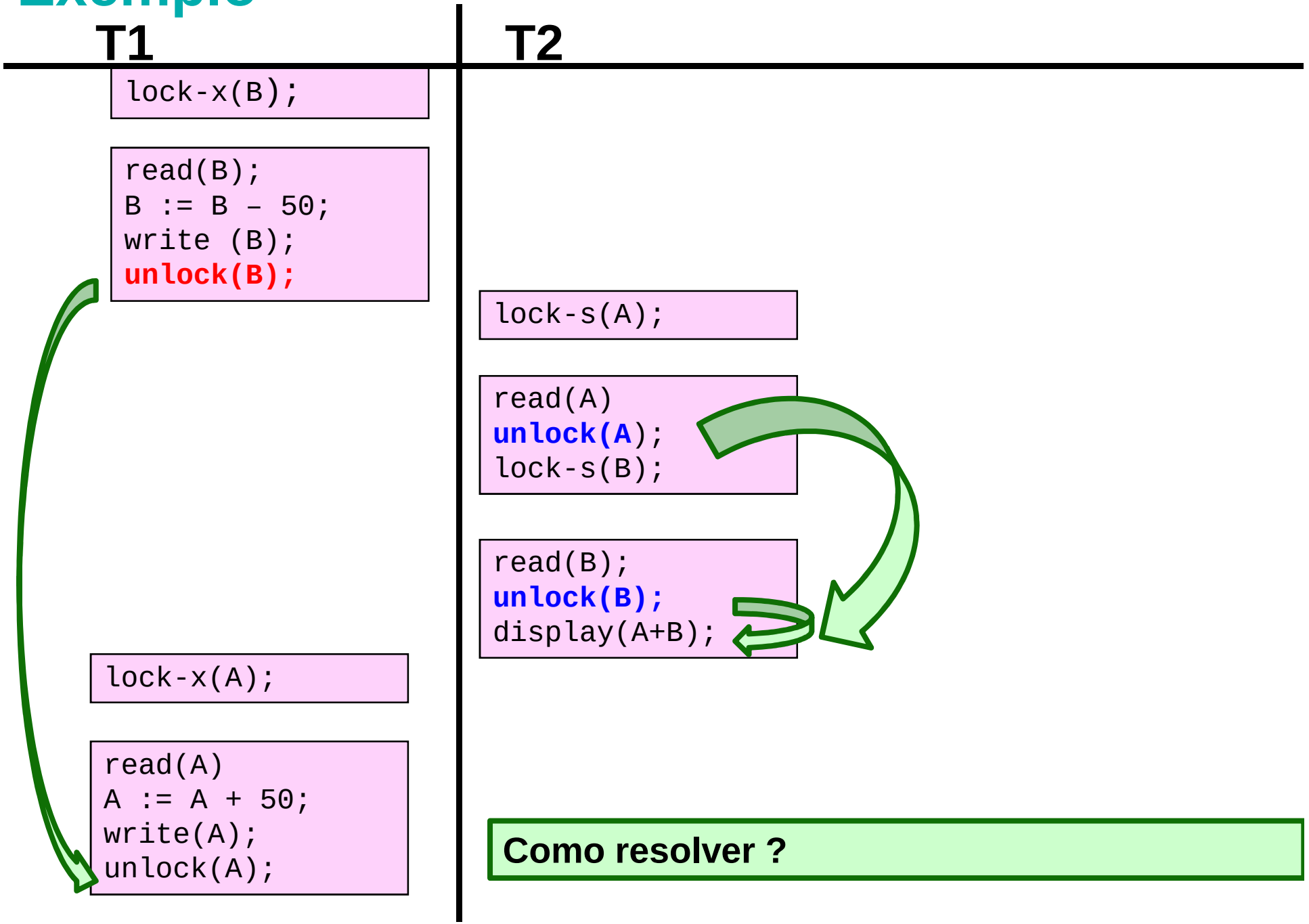
Supondo valor inicial de A e B = 100 e 200, respectivamente, que valor é mostrado por T2?

250! Por quê? T1 desbloqueou B muito cedo!

Exemplo


T1
T2
gerenciador de controle de concorrência
lock-x(B);
grant-x(B,T1)
read(B);
B := B – 50;
write (B);
unlock(B);
lock-s(A);
grant-s(A,T2)
read(A)
unlock(A);
lock-s(B);
grant-s(B,T2)
read(B);
unlock(B);
display(A+B);
lock-x(A);
grant-s(A,T1)
read(A)
A := A + 50;
write(A);
unlock(A);
Como resolver ?

Exemplo


T1
T2
lock-x(B);
read(B);
B := B – 50;
write (B);
unlock(B);
lock-s(A);
read(A)
unlock(A);
lock-s(B);
read(B);
unlock(B);
display(A+B);
lock-x(A);
read(A)
A := A + 50;
write(A);
unlock(A);
Como resolver ?

Exemplo

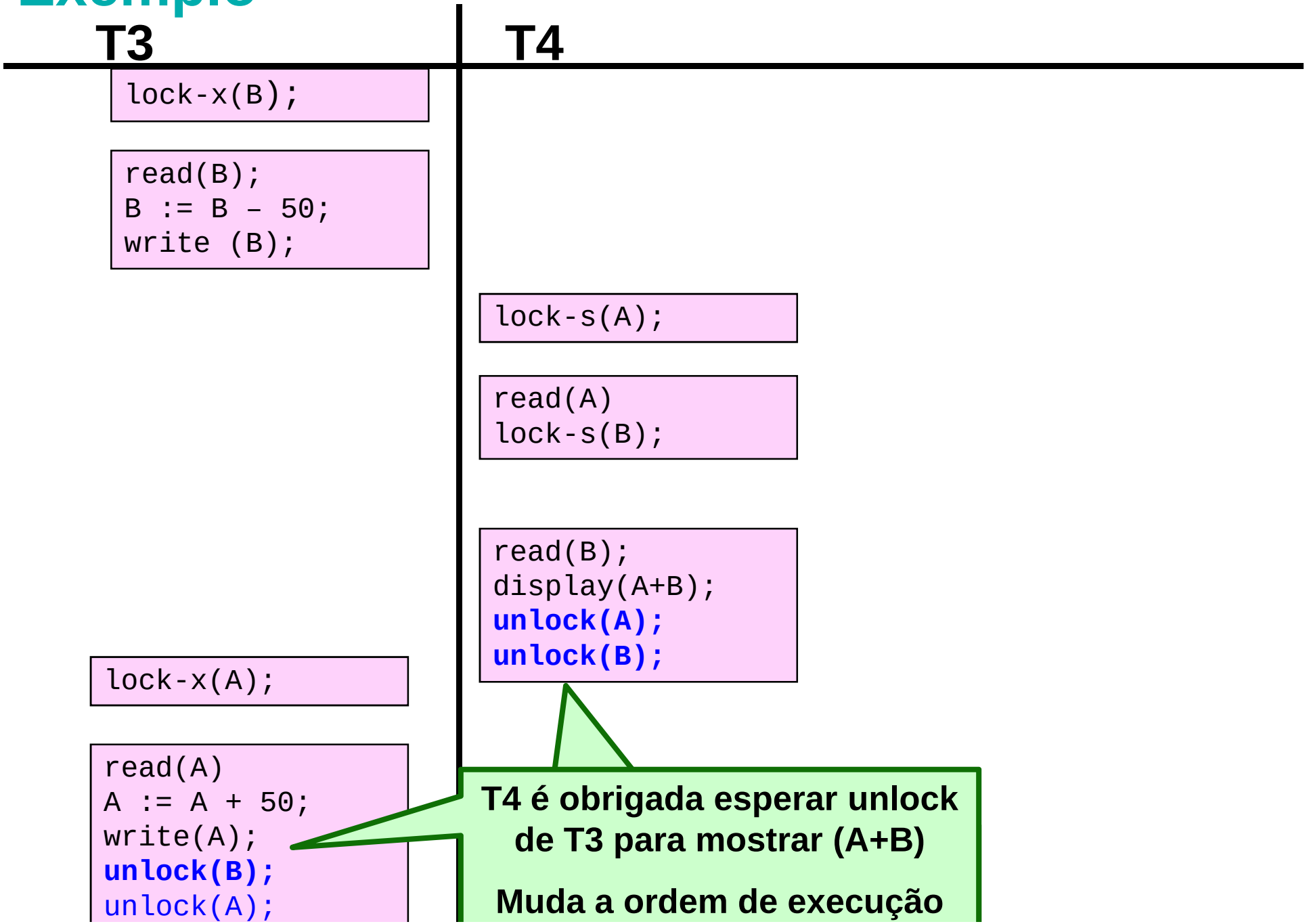
T3	T4
lock-x(B);	
read(B); B := B – 50; write (B);	
	lock-s(A);
	read(A) lock-s(B);

	read(B);
	display(A+B);
	unlock(A);
	unlock(B);

lock-x(A);	

read(A)
A := A + 50;
write(A);
unlock(B);
unlock(A);	

Exemplo


T3
T4
lock-x(B);
read(B);
B := B – 50;
write (B);
lock-s(A);
read(A)
lock-s(B);
read(B);
display(A+B);
unlock(A);
unlock(B);
lock-x(A);
read(A)
A := A + 50;
write(A);
unlock(B);
unlock(A);
T4 é obrigada esperar unlock de T3 para mostrar (A+B)
Muda a ordem de execução

Exemplo

T1	T2
`lock-x(B);`	

`read(B);`
`B := B – 50;`
`write (B);`	

	`lock-s(A);`

	`read(A)`
	`unlock(A);`
	`lock-s(B);`

`lock-x(A);`	

Algum problema neste schedule?

Exemplo

T1	T2
lock-x(B);	
read(B); B := B – 50; write (B);	
	lock-s(A);
	read(A) unlock(A); **lock-s(B);**
lock-x(A);	

Algum problema neste schedule?

Exemplo

T1	T2
lock-x(B);	
read(B); B := B – 50; write (B);	
	lock-s(A);
	read(A) unlock(A); **lock-s(B);**
lock-x(A);	

Algum problema neste schedule?

Impasse! DEAD-LOCK

Exemplo

T1	T2
lock-x(B);	
read(B); B := B – 50; write (B);	
	lock-s(A);
	read(A) unlock(A); **lock-s(B);**
lock-x(A);	

Algum problema neste schedule?

Impasse! DEAD-LOCK

Muitas vezes impasses são necessários para evitar inconsistências.

Sistema precisa reverter uma das transações para liberar os dados.

Protocolo de bloqueio

- ✓ Para evitar situações de impasse, cada transação deve seguir um conjunto de regras, chamadas **protocolo de bloqueio**.
  - ✓ protocolo indica quando uma transação pode bloquear e desbloquear um dado.
  - ✓ protocolos de bloqueio restringem o número de *schedules* possíveis.

Protocolo de bloqueio

- ✓ Definições para protocolo de bloqueio:
  - ✓ $\{T_0, T_1, ..., T_n\}$: conjunto de transações de um *schedule* S.
  - ✓ T_i precede T_j em S ($T_i \rightarrow T_j$):
    - se houver dado Q tal que T_i tenha mantido o modo de bloqueio A sobre Q e;
    - T_j tenha mantido o modo de bloqueio B sobre Q mais tarde e;
    - comp(A,B) = false.
  - ✓ Se $T_i \rightarrow T_j$, em qualquer *schedule* serial equivalente:
    - T_i precisa aparecer antes de T_j.

Protocolo de bloqueio

- ✓ Definições para protocolo de bloqueio:
- ✓ *Schedule* legal sobre determinado protocolo de bloqueio:
  - *schedule* possível para um conjunto de transações que seguem as regras do protocolo de bloqueio.
- ✓ *Schedule* assegura seriação de conflito se todos *schedules* legais forem passíveis de seriação de conflito.

Concessão de bloqueios

- ✓ Quando gerenciador pode conceder bloqueios?

Concessão de bloqueios

- ✓ Quando gerenciador pode conceder bloqueios?
  - ➢ quando transação solicita bloqueio em um modo sobre um dado e este dado não está bloqueado em modo incompatível;

Concessão de bloqueios

- ✓ Quando gerenciador pode conceder bloqueios?
  - ➢ quando transação solicita bloqueio em um modo sobre um dado e este dado não está bloqueado em modo incompatível;
  - ➢ mas deve-se evitar a situação:
    - T_2 tem bloqueio sobre Q no modo compartilhado
    - T_1 solicita bloqueio no modo exclusivo ➔ tem que esperar T_2 liberar
    - nesse meio de tempo T_3 solicita bloqueio compartilhado
    - solicitação de T_3 é compatível com bloqueio de T_2 ➔ bloqueio concedido
    - T_2 libera bloqueio, mas T_1 agora tem que T_3 liberar
    - se T_4 solicitar bloqueio compartilhado, situação se repete
    - T_1 é considerada **estagnada (starved)**

Concessão de bloqueios

- ✓ Como evitar estagnação (*starvation*)?

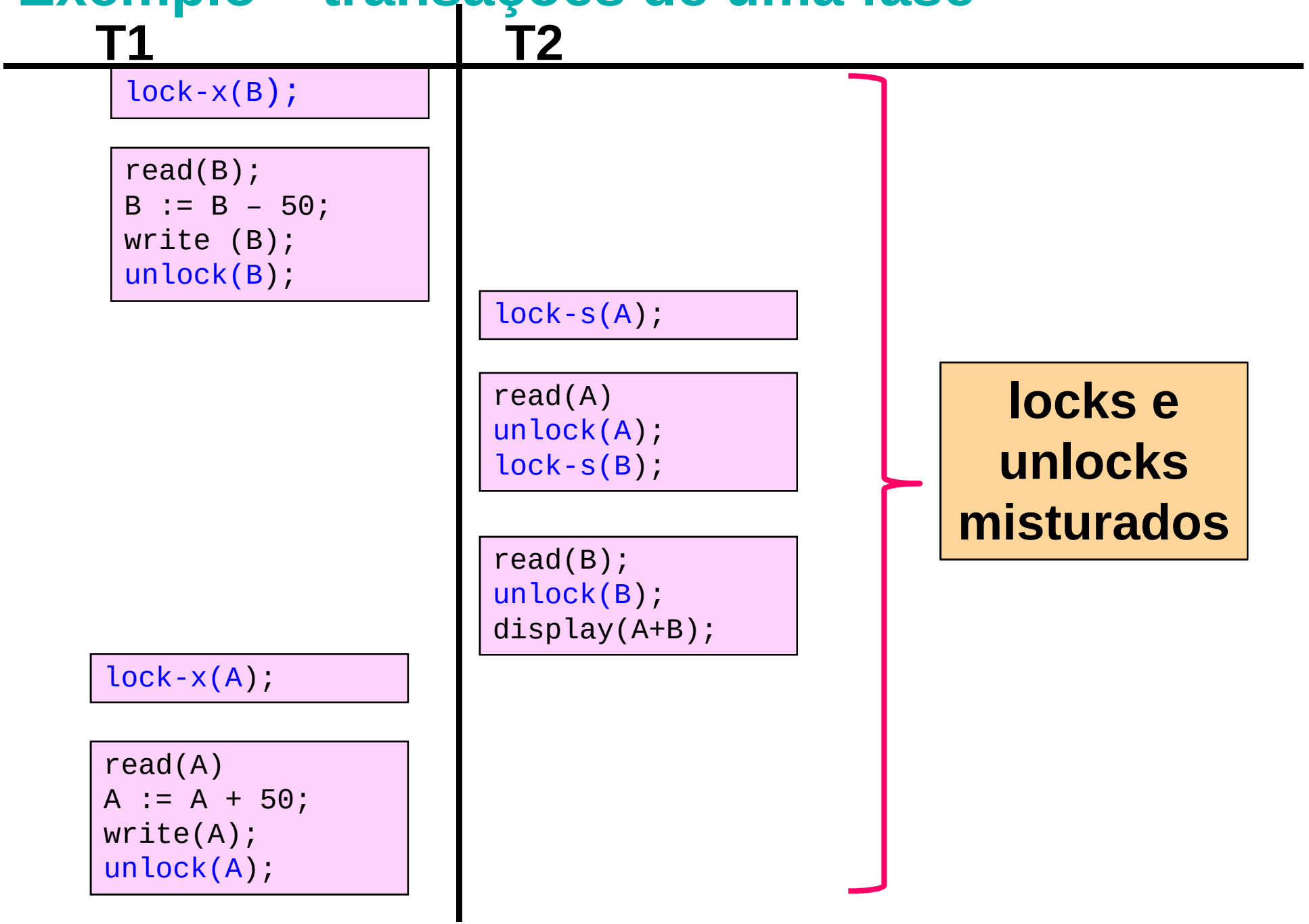
Concessão de bloqueios

- ✓ Como evitar estagnação (*starvation*)?
  - ➢ quando T_i solicita bloqueio sobre dado Q em um modo M, gerenciador de controle de concorrência concede bloqueio se:
    - não houver outra transação mantendo bloqueio sobre Q em um modo que entra em conflito com M;
    - não existir outra transação esperando por bloqueio sobre Q e que fez sua solicitação de bloqueio antes de T_i.

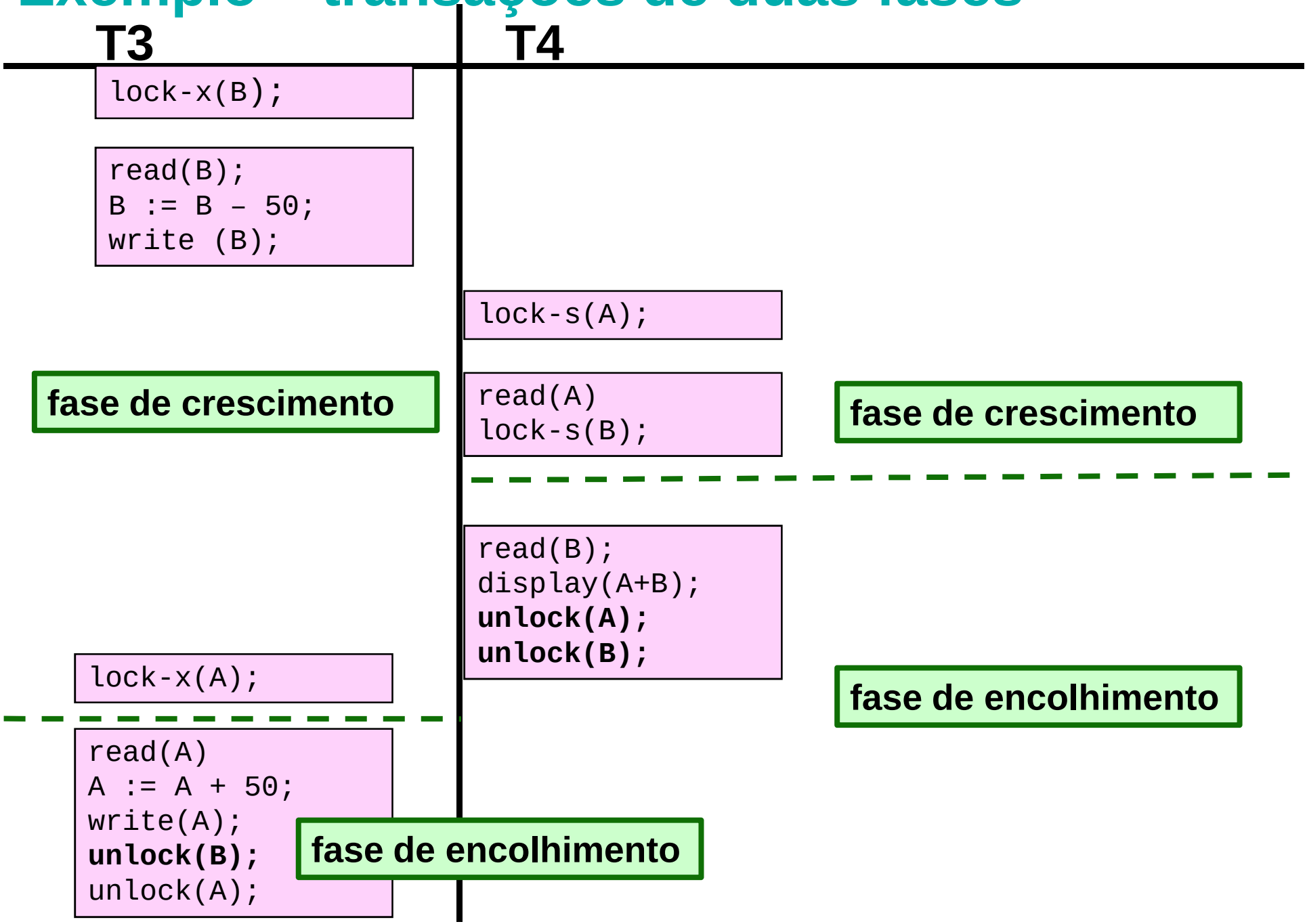
Protocolo de bloqueio em duas fases

- ✓ Tipo de protocolo que assegura a seriação
- ✓ Requer que cada transação emita solicitações de bloqueio e desbloqueio em duas fases:
  1. Fase de crescimento: transação pode obter bloqueios, mas não liberá-los.
  2. Fase de encolhimento: transação pode liberar bloqueios, mas não obter novos.
- ✓ Bloqueio em duas fases assegura seriação de conflito, mas não garante a liberdade do impasse.

Exemplo – transações de uma fase



Exemplo – transações de duas fases



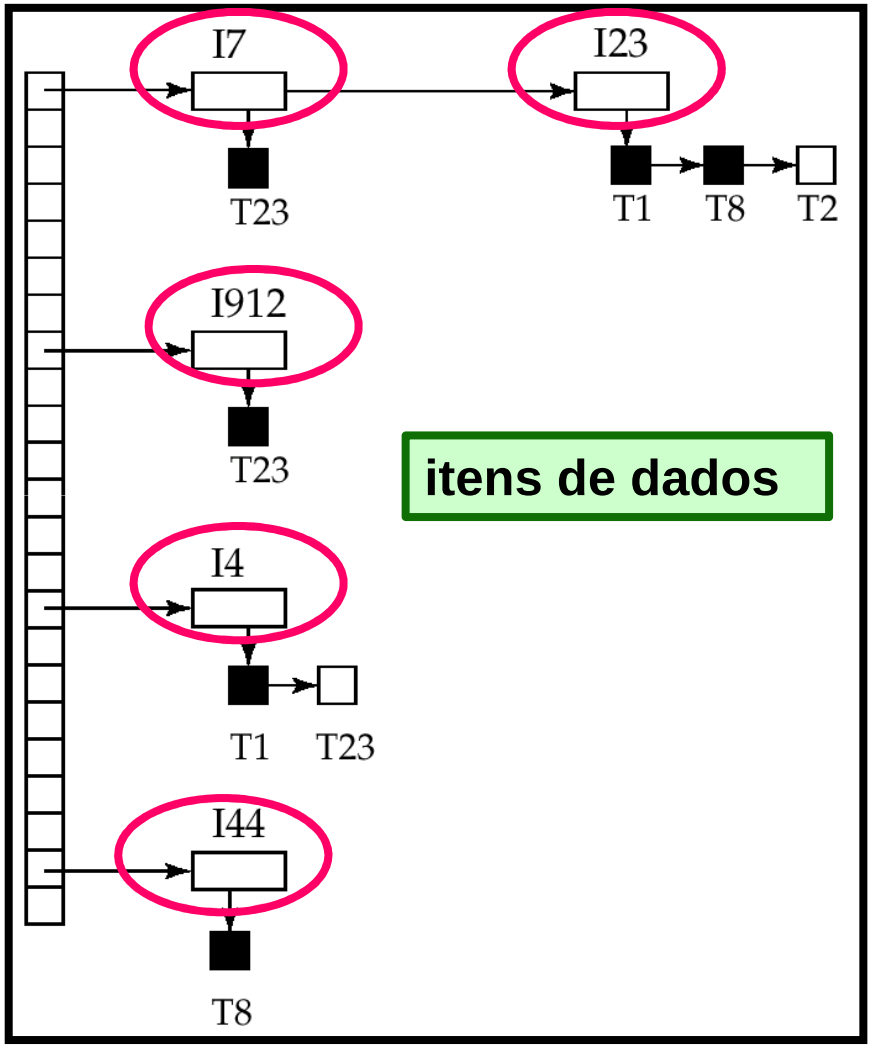
Implementação do bloqueio

- ✓ Como implementar um gerenciador de bloqueio?

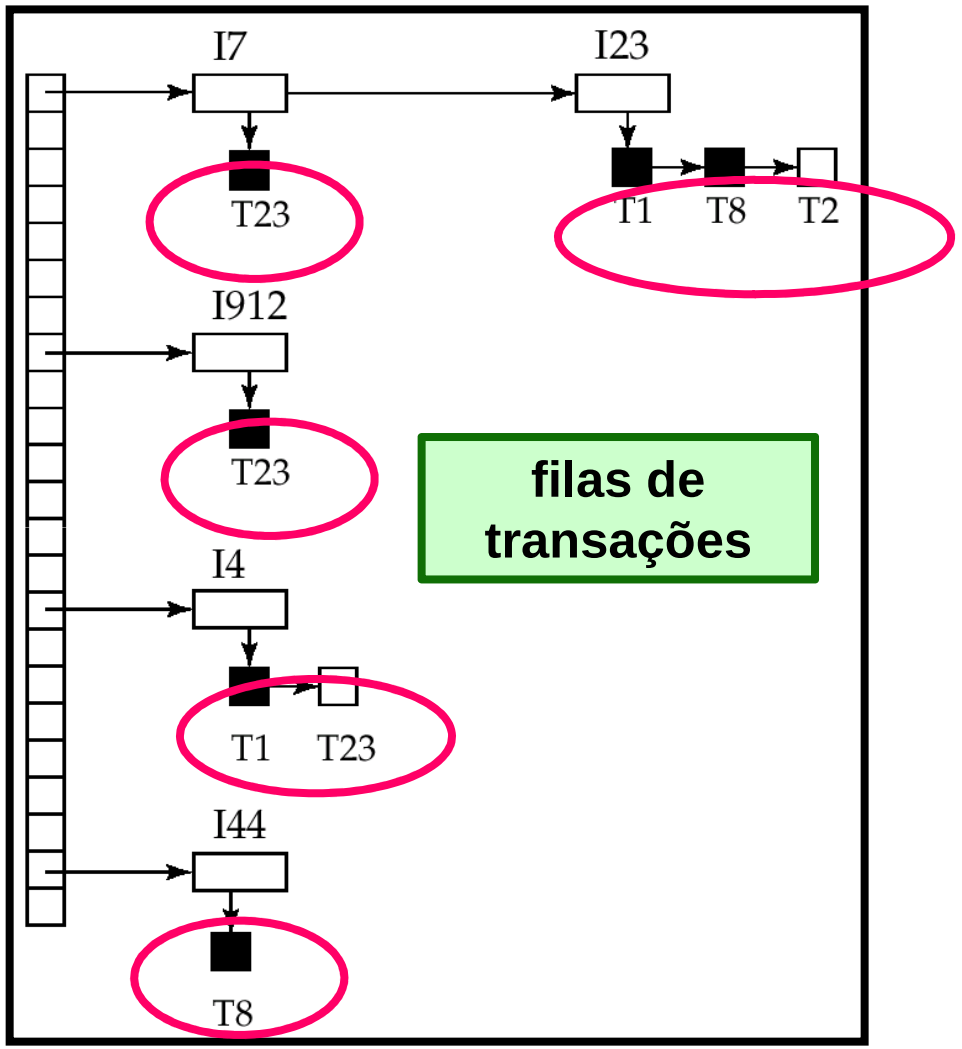
Implementação do bloqueio

- ✓ Como implementar um gerenciador de bloqueio?
  - ➢ processo que recebe mensagens de transações e envia mensagens em resposta;
  - ➢ responde mensagens de solicitação de bloqueio com mensagens de concessão de bloqueio ou solicitando rollback da transação;
  - ➢ mensagens de desbloqueio: apenas confirmação em resposta;
  - ➢ para cada item de dados atualmente bloqueado:
    - ➢ lista interligada de registros – um para cada solicitação
    - ➢ **tabela de bloqueio**: tabela hash indexada sobre o nome do item de dado.

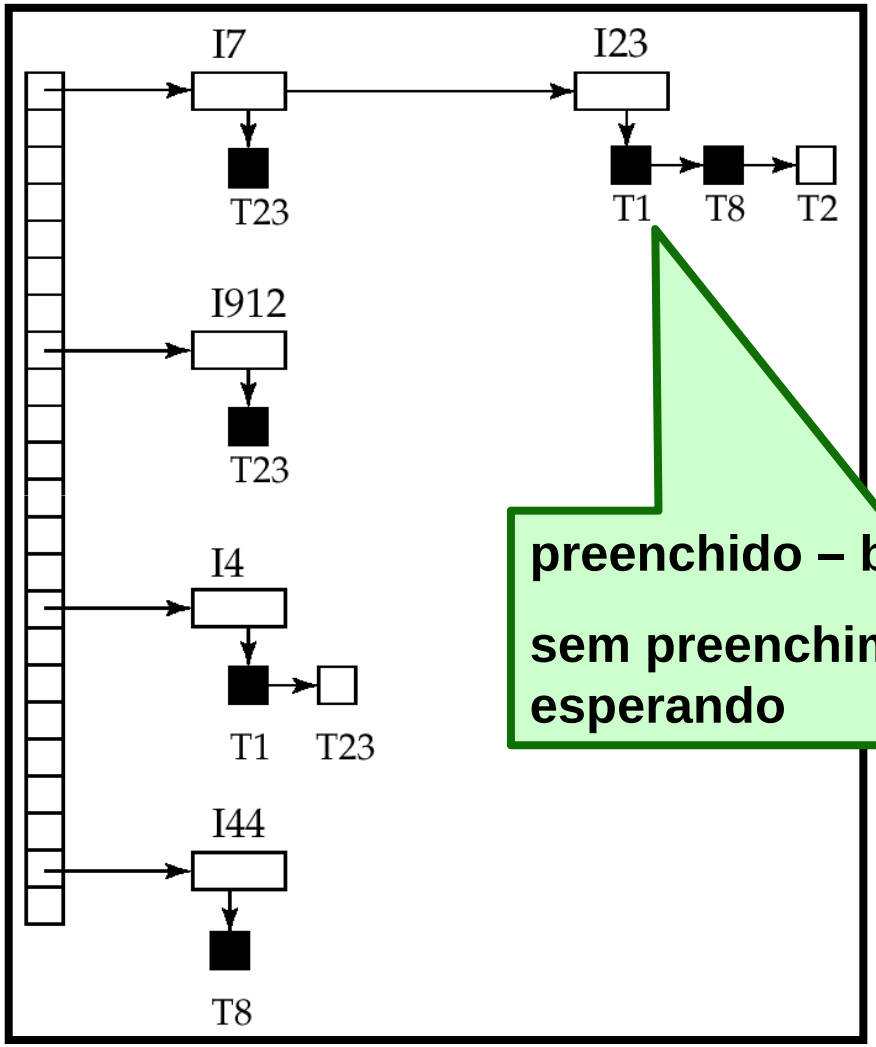
Implementação do bloqueio



Implementação do bloqueio



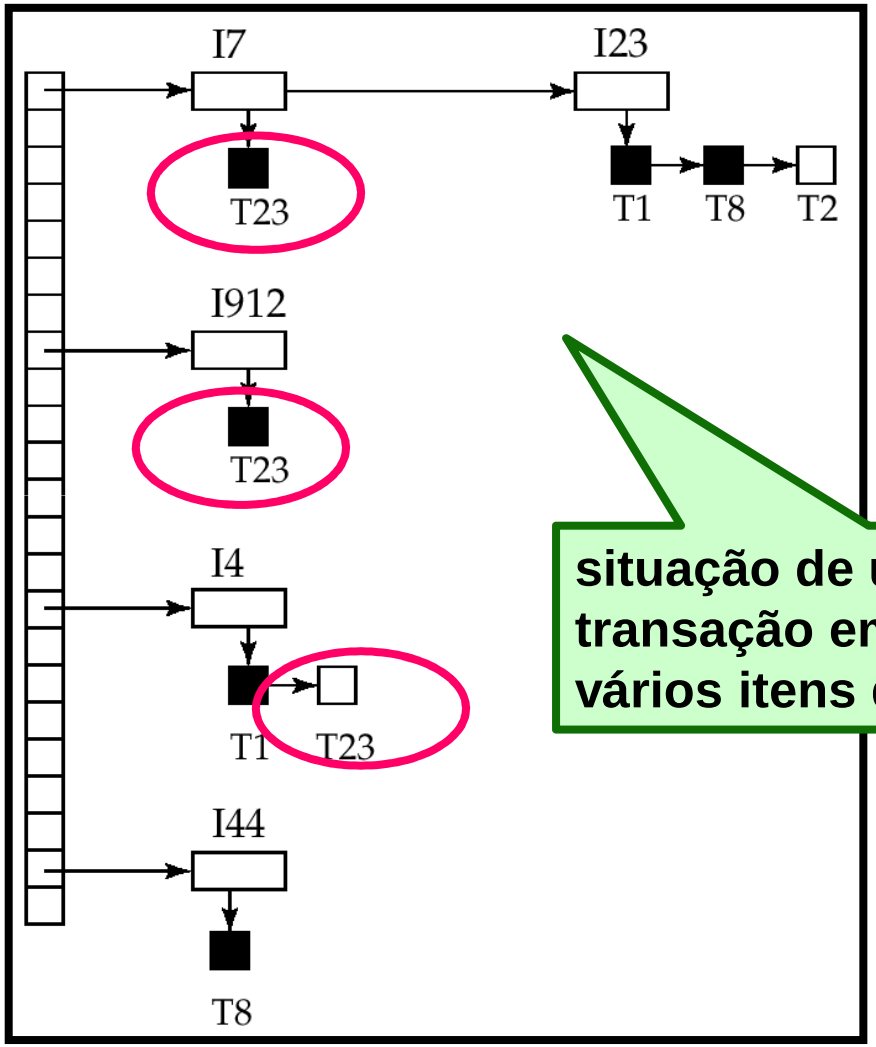
Implementação do bloqueio



preenchido – bloqueio concedido

sem preenchimento – bloqueio esperando

Implementação do bloqueio



situação de uma mesma transação em relação a vários itens de dado.

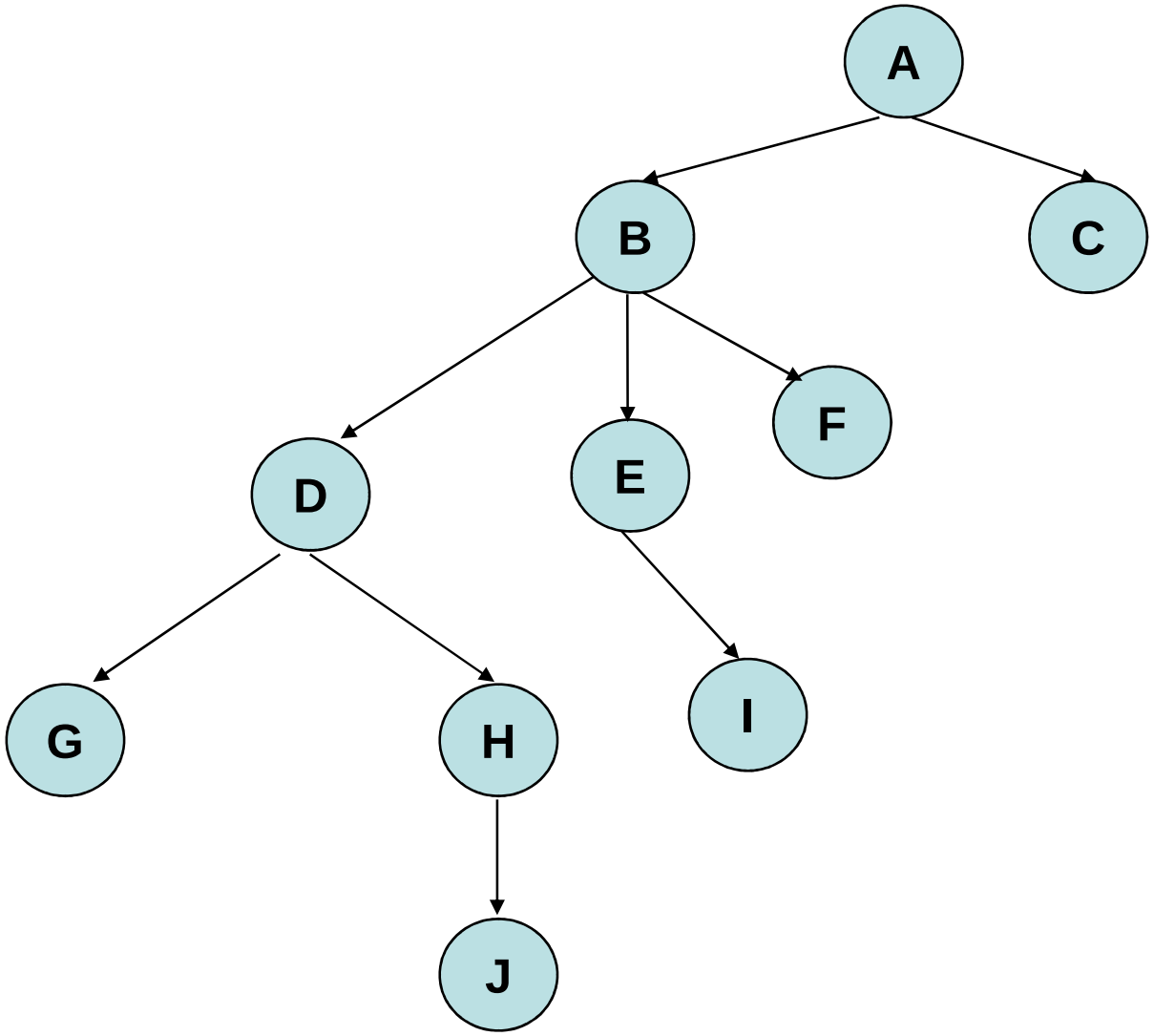
Protocolos baseados em gráficos

- ✓ Para implementar protocolos que não sejam em duas fases:
  - ✓ requer informações adicionais sobre como cada transação acessará o BD.
- ✓ Modelo mais simples:
  - ✓ conhecer ordem que os itens de dados serão acessados
  - ✓ isso garante seriação de conflito.
  - ✓ para isso: impor ordenação parcial sobre um conjunto de dados.

Protocolos baseados em gráficos

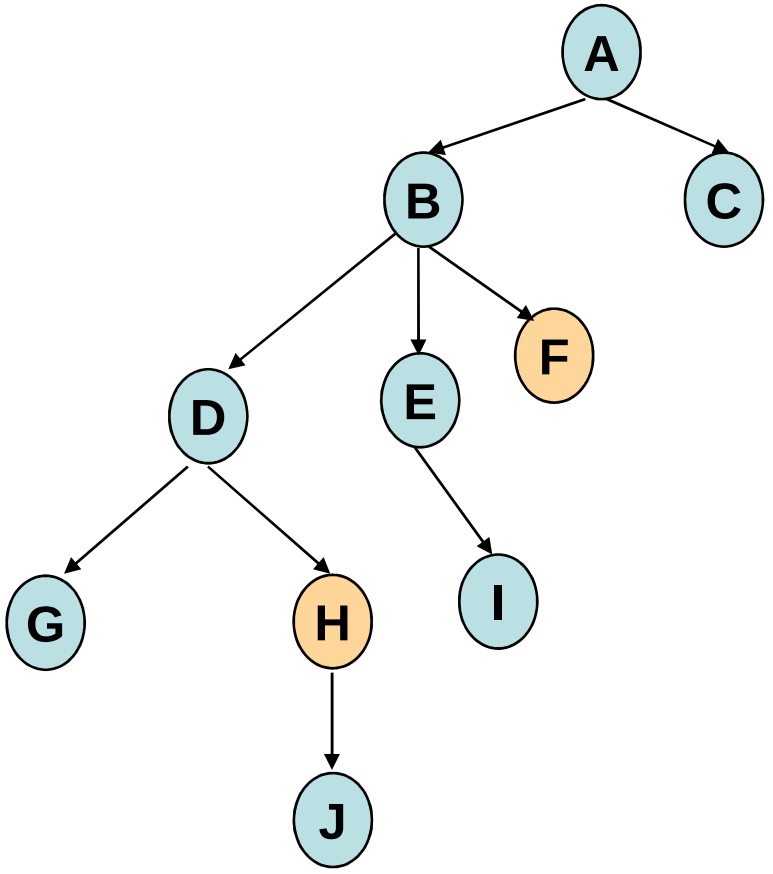
- ✓ Modelo mais simples:
  - ✓ Dado o conjunto de itens de dado D={d_1, d_2, ..., d_n}:
    - ✓ se temos ordenação parcial $d_i \rightarrow d_j$, qualquer transação precisa acessar d_i antes de d_j
    - ✓ ordenação parcial pode ser resultado da organização lógica ou física dos dados ou somente ser imposta para fins de controle de concorrência.
  - ✓ Ordenação do conjunto **D** pode ser vista como um gráfico acíclico direcionado: **gráfico de banco de dados**.
  - ✓ São definidos protocolos que trabalham com o gráfico formado. Ex: protocolo de árvore.

Protocolos baseados em gráficos



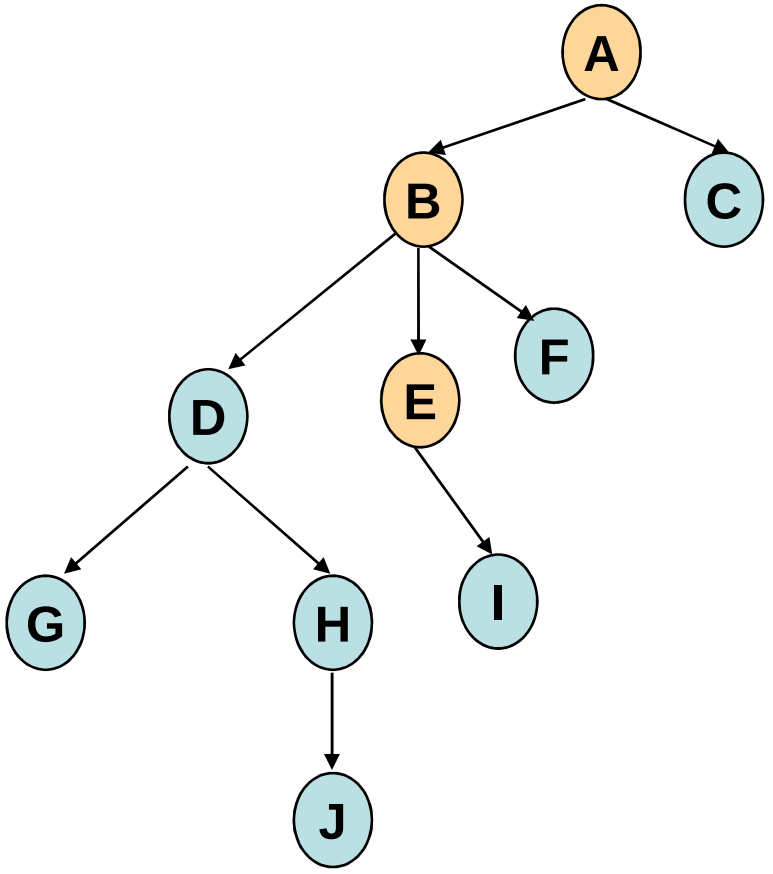
Protocolos baseados em gráficos

- ✓ **Protocolo de árvore:**
  - ✓ só permitida instrução **lock-x**;
  - ✓ regras para transação poder bloquear item de dados:
    - primeiro bloqueio pode ser sobre qualquer item;
    - item Q pode ser bloqueado por T_i somente se os pais de Q estiverem atualmente bloqueado por T_i;
    - itens de dados podem ser desbloqueados a qualquer momento;
    - item que foi bloqueado e desbloqueado por T_i não pode mais tarde ser bloqueado novamente por T_i.



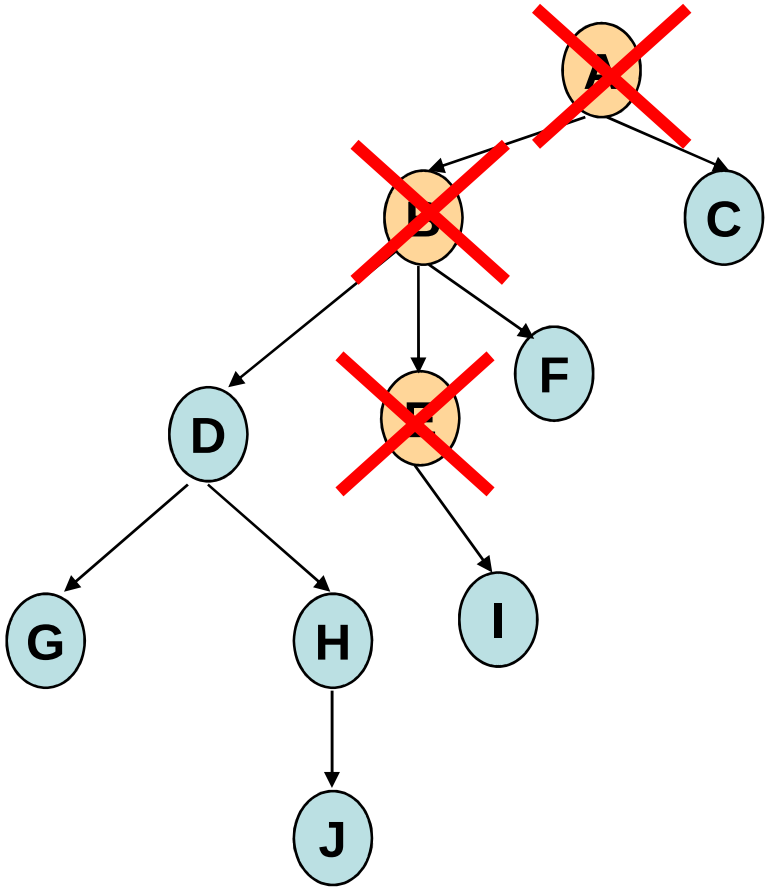
Protocolos baseados em gráficos

- ✓ Protocolo de árvore:
  - ✓ só permitida instrução **lock-x**;
  - ✓ regras para transação poder bloquear item de dados :
    - primeiro bloqueio pode ser sobre qualquer item;
    - item Q pode ser bloqueado por T_i somente se os pais de Q estiverem atualmente bloqueado por T_i;
    - itens de dados podem ser desbloqueados a qualquer momento;
    - item que foi bloqueado e desbloqueado por T_i não pode mais tarde ser bloqueado novamente por T_i.



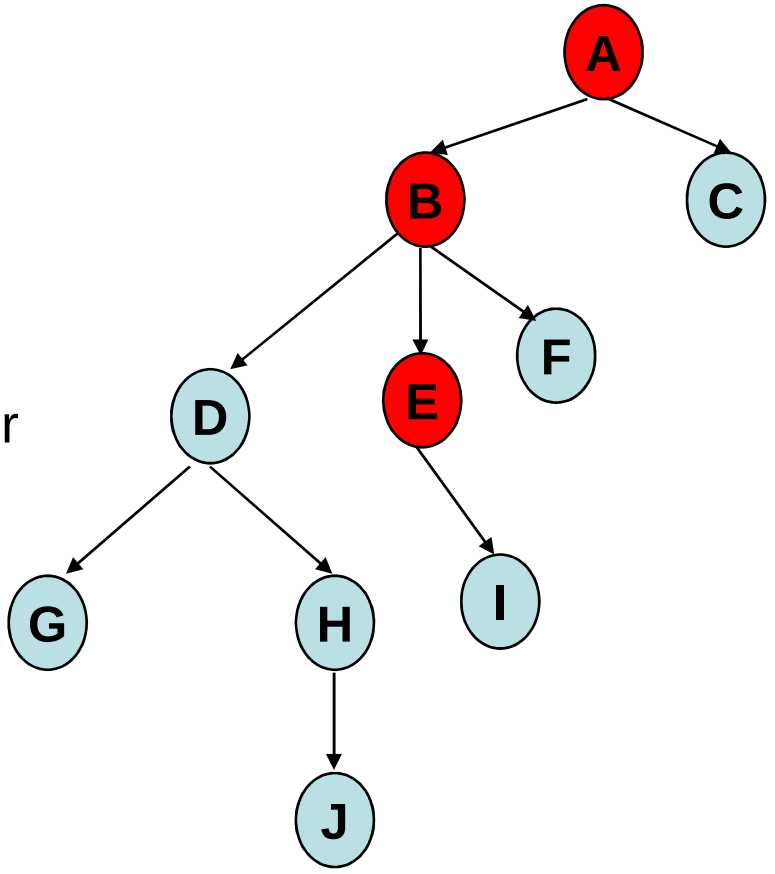
Protocolos baseados em gráficos

- ✓ Protocolo de árvore:
  - ✓ só permitida instrução **lock-x**;
  - ✓ regras para transação poder bloquear item de dados :
    - primeiro bloqueio pode ser sobre qualquer item;
    - item Q pode ser bloqueado por T_i somente se os pais de Q estiver atualmente bloqueado por T_i;
    - itens de dados podem ser desbloqueados a qualquer momento;
    - item que foi bloqueado e desbloqueado por T_i não pode mais tarde ser bloqueado novamente por T_i.



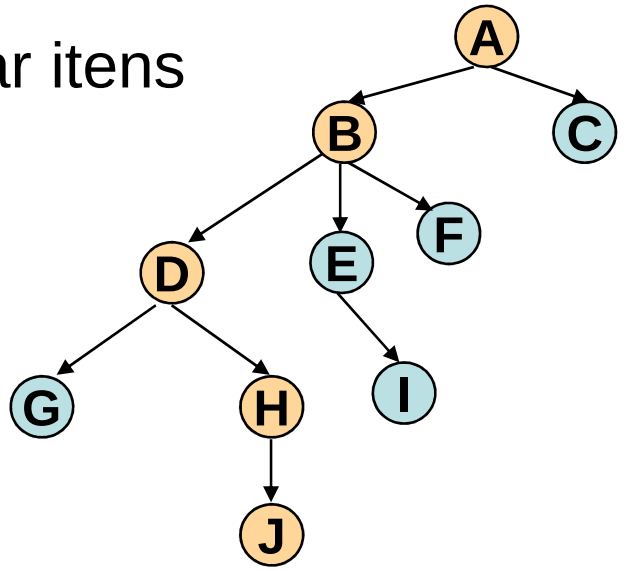
Protocolos baseados em gráficos

- ✓ Protocolo de árvore:
  - ✓ só permitida instrução **lock-x**;
  - ✓ regras para transação poder bloquear item de dados :
    - primeiro bloqueio pode ser sobre qualquer item;
    - item Q pode ser bloqueado por T_i somente se o pai de Q estiver atualmente bloqueado por T_i;
    - itens de dados podem ser desbloqueados a qualquer momento;
    - item que foi bloqueado e desbloqueado por T_i não pode mais tarde ser bloqueado novamente por T_i.



Protocolos baseados em gráficos

- ✓ Protocolo de árvore:
  - ✓ vantagens sobre protocolo em duas fases:
    - livre de impasse – nenhum *rollback* é exigido;
    - desbloqueio pode ocorrer mais cedo – tempos de espera menores e aumento de concorrência.
  - ✓ desvantagens:
    - transação pode ter que bloquear itens de dados que ela não acessa – sobrecarga de bloqueio.
      Exemplo: transação tentando acessar itens A e J – precisa bloquear também B, D e H.



Protocolos baseados em *Timestamp*

- ✓ *Timestamp*: identificador único criado pelo SGBD para identificar uma transação (clock do sistema ou contador). São associados na ordem em que as transações são submetidas ao sistema.
  - TS(T) = *timestamp* da transação T
- ✓ Protocolo:
  - ✓ garante seriação;
  - ✓ uso de *timestamp* para ordenar a execução das transações de forma que o escalonamento formado seja equivalente ao escalonamento serial;
  - ✓ no escalonamento as transações usam a ordem de seus *timestamps;*
  - ✓ Se $TS(T_i) < TS(T_j)$, o sistema garante que o escalonamento produzido é equivalente ao escalonamento serial $<T_i, T_j>$.

Protocolos baseados em *Timestamp*

✓Ações do algoritmo de ordenamento por *timestamp:*

- checa se duas operações conflitantes ocorrem na ordem incorreta. Se ocorrerem, rejeita a última das duas operações abortando a transação que errou;
- garantia da seriação por conflito;
- sempre que uma transação tenta ler ou escrever um item de dado Q, o algoritmo compara o *timestamp* dela com o *timestamp* de leitura ou escrita de Q;
- intenção: assegurar que a ordenação *timestamp* não seja violada;
- se a ordenação *timestamp* for violada, a transação será abortada.

Protocolos baseados em *Timestamp*

- ✓ Funções do algoritmo básico:
  - ✓ Associa dois *timestamps* a cada item Q:
    - **R-timestamp(Q):** o *timestamp* de leitura de Q é o maior entre todos os *timestamps* de transações que leram Q com sucesso;
    - **W-timestamp(Q):** o *timestamp* de escrita de Q é o maior entre todos os *timestamps* de transações que escreveram Q com sucesso;
    - *timestamps* são atualizados sempre que uma instrução read(Q) ou write(Q) é executada;
    - Assegura que as operações de leitura e escrita sejam executadas por ordem de *timestamp.*

Protocolos baseados em *Timestamp*

✓ Situação 1: a transação T_i emite read(Q).

- Se TS(T_i) < W-timestamp(Q), então T_i quer ler o valor de Q que já foi modificado por uma transação mais nova. Assim, a operação read é rejeitada e T_i é desfeita.

- Se TS(T_i) ≥ **W-timestamp(Q)**, a transação que requer o dado é mais nova (ou é a própria) que a última que o escreveu. Assim, a operação read é executada e o **R-timestamp(Q)** recebe o maior valor entre o atual e o TS(T_i).

Protocolos baseados em *Timestamp*

✓ Situação 2: a transação T_i emite write(Q)

- Se $TS(T_i) <$ R-timestamp(Q), a transação é mais velha que a última transação que leu, ou seja, o valor de Q que T_i está produzindo seria necessário antes. A operação de write é rejeitada e T_i é desfeita.
- Se $TS(T_i) <$ W-timestamp(Q) então T_i está tentando escrever um valor obsoleto em Q. Logo, essa operação é rejeitada e T_i é desfeita.
- Caso contrário, a operação write é executada e o W-timestamp(Q) é atualizado como $TS(T_i)$.

Protocolos baseados em *Timestamp*

- Se uma transação foi revertida: mais tarde, a transação é resubmetida ao sistema como uma nova transação e com um novo *timestamp*.
- Se T_i é abortada e desfeita, qualquer transação que tenha usado um valor escrito por T_i deve ser desfeita também.
- Pode cair em *starvation.*

Exemplo – *timestamp*

T1	T2
read(B) read(A) display(A+B)	read(B) B := B - 50 write(B) read(A) A := A + 50 write(A) display(A+B)

Exemplo – *timestamp*

T1	T2
read(B)	
	read(B) B := B – 50 write(B)
read(A)	
	read(A)
display(A+B)	
	A := A + 50 write(A) display(A+B)

TS(T1) = 1 **TS(T2) = 2**

R-TS(B)	W-TS(B)	R-TS(A)	W-TS(A)	Op.
0	0	0	0	$R_1(B)$
1				$R_2(B)$
2				$W_2(B)$
	2			$R_1(A)$
		1		$R_2(A)$
		2		$W_2(A)$
			2	

Exemplo – *timestamp*

T1	T2
read(B)	
	read(B) B := B – 50 write(B)
read(A)	
	read(A)
display(A+B)	
	A := A + 50 write(A) display(A+B)
write(A)	

TS(T1) = 1 **TS(T2) = 2**

R-TS(B)	W-TS(B)	R-TS(A)	W-TS(A)	Op.
0	0	0	0	$R_1(B)$
1				$R_2(B)$
2				$W_2(B)$
	2			$R_1(A)$
		1		$R_2(A)$
		2		$W_2(A)$
			2	$W_1(A)$
			1	

Exemplo – *timestamp*

T1	T2
read(B)	
	read(B) B := B – 50 write(B)
read(A)	
	read(A)
display(A+B)	
	A := A + 50 write(A) display(A+B)

TS(T1) = 1 **TS(T2) = 2**

R-TS(B)	W-TS(B)	R-TS(A)	W-TS(A)	Op.
0	0	0	0	$R_1(B)$
1				$R_2(B)$
2				$W_2(B)$
	2			$R_1(A)$
		1		$R_2(A)$
		2		$W_2(A)$
			2	$W_1(A)$
			1	

$TS(T_i) < R\text{-timestamp}(A)$: transação é mais velha que a última transação que leu, ou seja, o valor de Q que T_i está produzindo seria necessário antes. A operação de write é rejeitada e T_i é desfeita.

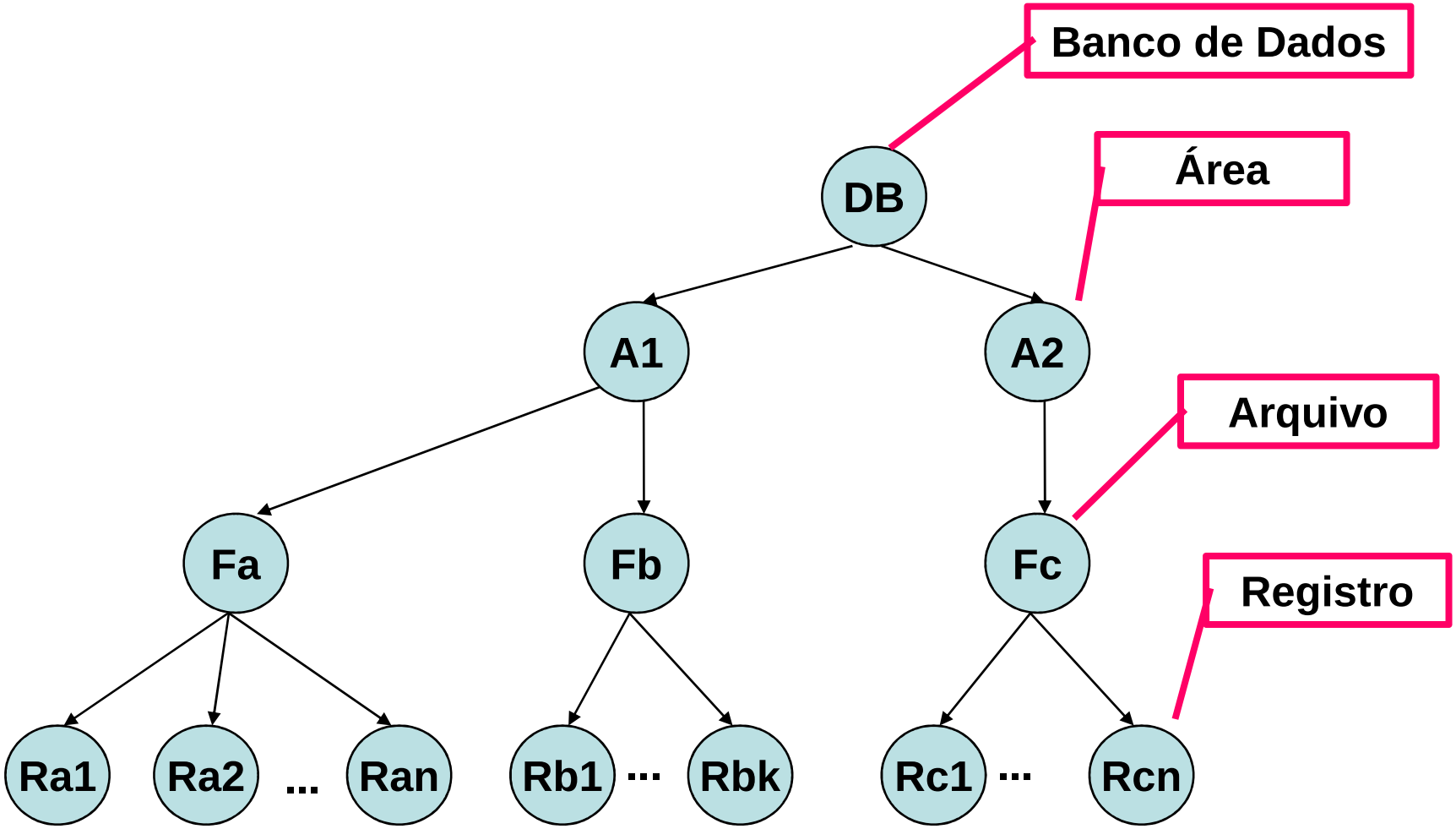
Regra de Thomas

- ✓ Modificação no algoritmo básico, que faz com ele rejeite menos operações de escrita, aumentado as possibilidades de concorrência.
- ✓ Deixa de garantir a seriação em conflito
- ✓ Alteração na regra 2:
  - ✓ Se $TS(T_i) <$ W-timestamp(Q) , então T_i está tentando escrever um valor obsoleto para Q. A operação de write pode ser ignorada.

Granularidade múltipla

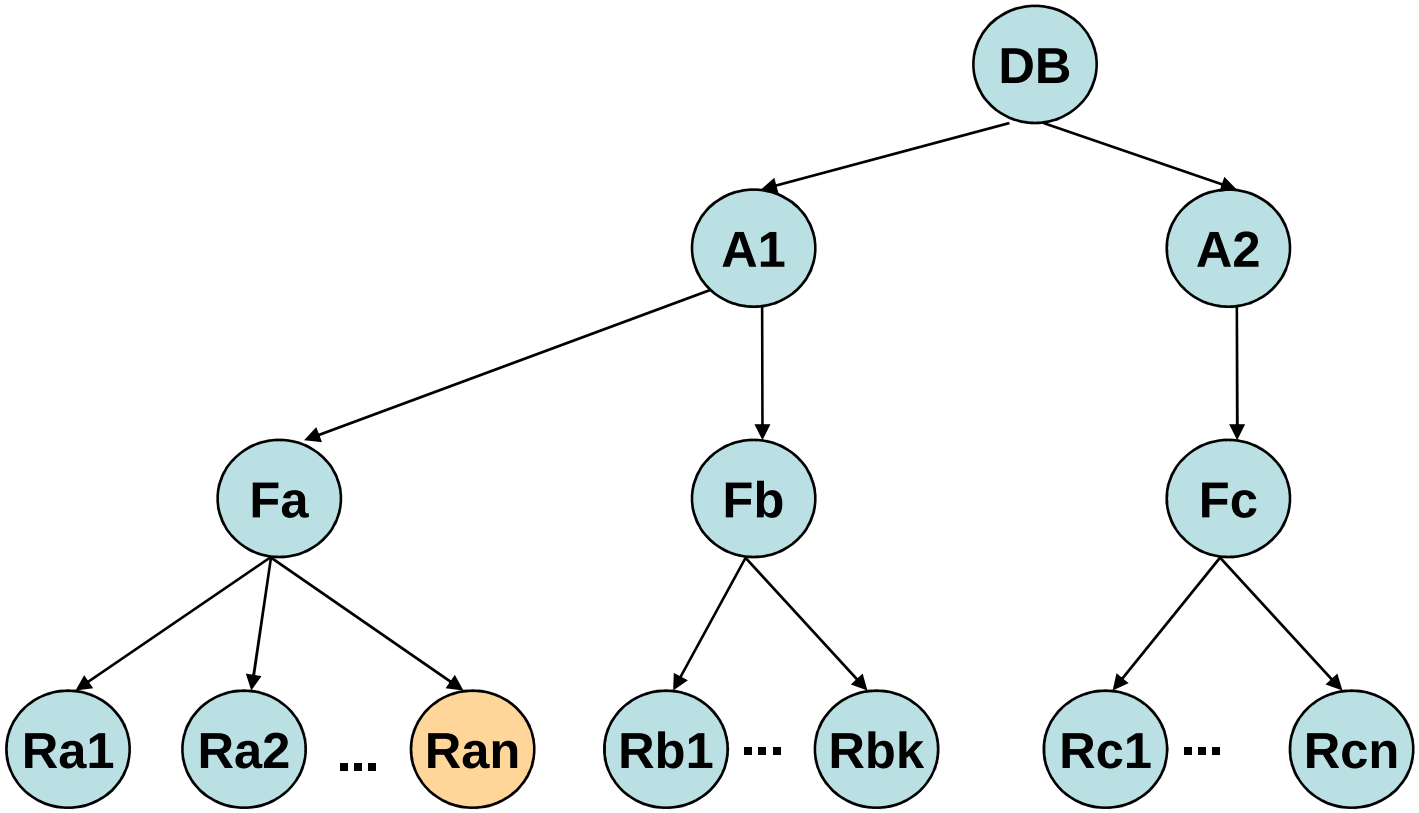
- ✓ Nos protocolos vistos, cada item de dado é a unidade de bloqueio ➔ demora.
- ✓ Solução: transação emitir solicitação para bloquear banco de dados inteiro ➔ BD indisponível para outras transações.
- ✓ Granularidade: mecanismo para permitir que itens de dados sejam de tamanhos variáveis, criando uma hierarquia em árvore.
  - ✓ cada nó pode ser bloqueado individualmente: **bloqueio explícito**;
  - ✓ se pai for bloqueado, seus filhos também o serão: **bloqueio implícito**.

Granularidade múltipla



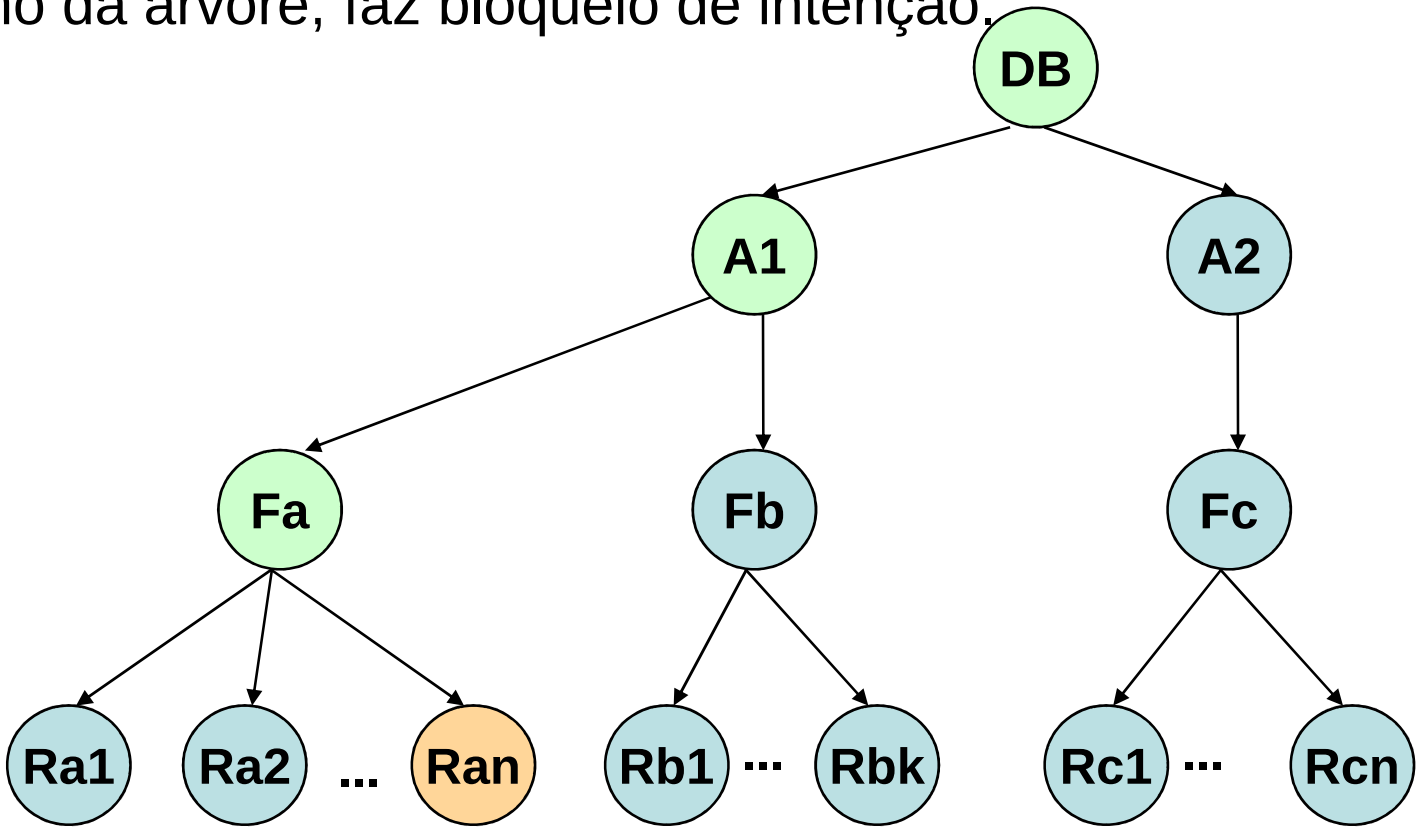
Granularidade múltipla

- ✓ Se uma transação quer bloquear um nó, como sabe se ele já está bloqueado?



Granularidade múltipla

- ✓ Se uma transação quer bloquear um nó, como sabe se ele já está bloqueado?
  - ✓ Solução 1: bloquear BD todo: inviável
  - ✓ Solução: **bloqueio intencional** – enquanto percorrer um caminho da árvore, faz bloqueio de intenção.



Granularidade múltipla

- ✓ Função de compatibilidade do bloqueio intencional

	IS	IX	S	SIX	X
IS	V	V	V	V	F
IX	V	V	F	F	F
S	V	F	V	F	F
SIX	V	F	F	F	F
X	F	F	F	F	F

IS = intencional compartilhado
IX = intencional exclusivo
S = compartilhado
SIX = compartilhado e exclusivo intencional
X = exclusivo

Granularidade múltipla

- ✓ Protocolo de bloqueio
  - garante serialização;
  - transação T_i pode bloquear um nó Q, usando as seguintes regras:
    1. observar função de compatibilidade de bloqueio;
    2. bloquear primeiro raiz da árvore (pode ser bloqueada em qualquer modo);
    3. nó Q pode ser bloqueado por T_i no modo S ou IS somente se o seu pai for bloqueado por T_i no modo IX ou IS;
    4. nó Q pode ser bloqueado por T_i no modo X, SIX ou IX somente se o seu pai estiver bloqueado por T_i no modo IX ou SIX;
    5. T_i pode bloquear um nó somente se ele não desbloqueou outro nó anteriormente (isto é, T_i tem duas fases);
    6. T_i pode desbloquear um nó Q somente se nenhum dos filhos de Q estiver bloqueado por T_i.

Granularidade múltipla

- ✓ Protocolo de bloqueio:
  - aumenta a concorrência e reduz o *overhead* por bloqueios;
  - bloqueios devem ser adquiridos no sentido raiz → folha e desbloqueio no sentido contrário;
  - útil para aplicações com:
    - transações curtas que mantêm acesso a poucos itens de dados;
    - transações longas que produzem relatório a partir de um arquivo ou de um conjunto de arquivos.

Esquemas de múltipla versão

- ✓ Esquemas de controle de concorrência por bloqueio ou por *timestamp* garantem a serialização atrasando a operação ou abortando a transação responsável por tal operação.
- ✓ Outra forma de fazer: sistema providenciar cópias de cada item de dado.
- ✓ Sistema de banco de dados *multiversão:*
  - cada operação **write(Q)** cria uma nova versão de Q;
  - quando é emitida uma operação **read(Q),** o sistema seleciona uma das versões de Q para ser lida.
- ✓ Controle de concorrência precisa garantir que a seleção da versão lida seja tal que assegure a seriação.
- ✓ Por razões de desempenho, transação precisa determinar de forma fácil e rápida qual versão do item de dados que deverá ser lida.

Ordenação por *timestamp* em múltipla versão

- ✓ Para cada transação T_i do sistema é associado um *timestamp* único e estático, denotado por $TS(T_i)$ ➔ associado antes do início da execução da transação.

- ✓ Para cada item de dado Q é associada uma sequência de versões $<Q_1, Q_2, ..., Q_n>$. Cada versão Q_k contém três atributos:
  - Conteúdo: é o valor da versão Q_k;
  - W-timestamp(Q_k): é o *timestamp* da transação que criou a versão Q_k.
  - R-timestamp(Q_k): é o *timestamp* mais alto de alguma transação que tenha lido a versão Q_k com sucesso.

Ordenação por *timestamp* em múltipla versão

- ✓ Uma transação T_i cria uma nova versão Q_k do item de dado Q emitindo uma operação write(Q).
- ✓ O campo **conteúdo** da versão mantém o valor escrito por T_i.
- ✓ O W-timestamp e o R-timestamp são inicializados por $TS(T_i)$.
- ✓ O valor de R-timestamp é atualizado sempre que uma transação T_j lê o conteúdo de Q_k e $R\text{-timestamp}(Q_k) < TS(T_j)$.

Ordenação por *timestamp* em múltipla versão

- ✓ Considerando Q_k como a versão de Q cujo *timestamp* de escrita seja o maior *timestamp* de escrita menor ou igual a $TS(T_i)$, esquema assegura seriação, usando duas regras:

1. Se a transação T_i emitir um read(Q), então o valor retornado é o conteúdo da versão Q_k.
   - Isso faz com que uma transação leia a versão mais recente que vem antes dela no tempo.
2. Se a transação T_i emitir write(Q) e $TS(T_i) < R\text{-timestamp}(Q_k)$, o sistema reverte a transação T_i.
   Se $TS(T_i) = W\text{-timestamp}(Q_k)$, o sistema escreve sobre o conteúdo de Q_k; senão ele cria uma nova versão de Q.
   - Essa regra força uma transação a abortar se ela estiver muito atrasada ao fazer uma escrita.

Tratamento de impasse

- ✓ Sistema está em estado de impasse (*deadlock*) se houver conjunto de transações tal que cada transação esteja esperando por outra do conjunto.
- ✓ Dois métodos para tratamento de impasse:
  - prevenção de impasse: garantir que o sistema nunca entre em estado de impasse. Usada se probabilidade do sistema entrar em impasse for relativamente alta.
  - detecção e recuperação de impasse: permite que o sistema entre em um estado de deadlock e então remove-o deste estado, recuperando-o.

Tratamento de impasse - prevenção

- ✓ Técnica 1: obriga que cada transação bloqueie todos os itens de dados antes de sua execução. Ou todos são bloqueados de uma vez ou nenhum o será.
  - Desvantagens:
    - Dificuldade em prever, antes da transação começar, quais itens de dados precisarão ser bloqueados (seria necessário o uso de um esquema de informação nas transações);
    - A concorrência é bastante reduzida.

Tratamento de impasse - prevenção

- ✓ Técnica 2: preempção (apropriação) e *rollback* de transações.
  - Preempção: quando uma transação T2 solicita um bloqueio que está sendo mantido pela transação T1, o bloqueio concedido a T1 pode ser revertido com *rollback* de T1 e concedido a T2.
  - Para controlar esta preempção, considera-se um único *timestamp* para cada transação. Eles são usados para decidir se a transação pode esperar ou será revertida.
  - Bloqueio continua sendo usado para controlar o concorrência.
  - Se uma transação for revertida, ela manterá seu *timestamp* **antigo** quando for reiniciada.

Tratamento de impasse - prevenção

✓ Técnica 2: preempção (apropriação) e *rollback* de transações.

– Esquemas usando *timestamps:*

- **esperar-morrer**:
  - tem por base uma técnica de não-preempção.
  - quando uma transação T_i solicita um item de dado mantido por T_j,T_i pode esperar somente se possuir um *timestamp* menor do que T_j (isto é, T_i é mais antiga que T_j).
  - caso contrário, T_i será revertida (morta).
- **ferir-esperar:**
  - tem por base a técnica da preempção e é uma contrapartida ao esquema esperar-morrer.
  - quando uma transação T_i solicita um item de dado mantido por T_j, T_i poderá esperar somente se possuir um *timestamp* maior que T_j (ou seja, T_i é mais nova que T_j).
  - caso contrário, T_j será desfeita (T_j é ferida por T_i).

Tratamento de impasse - prevenção

- ✓ Técnica 2: preempção (apropriação) e *rollback* de transações.
  - Diferença entre os esquemas:
    - esperar-morrer: transação mais antiga precisará esperar até que a mais nova libere seus itens de dados. Quanto mais antiga a transação, maior a possibilidade de esperar;
    - ferir-esperar: a transação mais antiga nunca espera a mais nova.
    - esperar-morrer: se uma transação T_i morre e é desfeita porque solicitou um item de dados preso por uma transação T_j, T_i pode reemitir a mesma sequência de solicitações quando for reiniciada. Se os itens de dados ainda estiverem presos por T_j, T_i morrerá novamente. T_i poderá morrer diversas vezes antes de conseguir o item de dados necessário.
    - ferir-esperar: a transação T_i será ferida e revertida porque T_j solicitou um item de dados preso por ela. Quanto T_i reinicia e solicita o item de dados preso por T_j, T_i esperará. Com isso, deve haver menos reversões.
  - Nos dois esquemas: maior problema = *rollbacks* desnecessários.

Tratamento de impasse – tempo limite

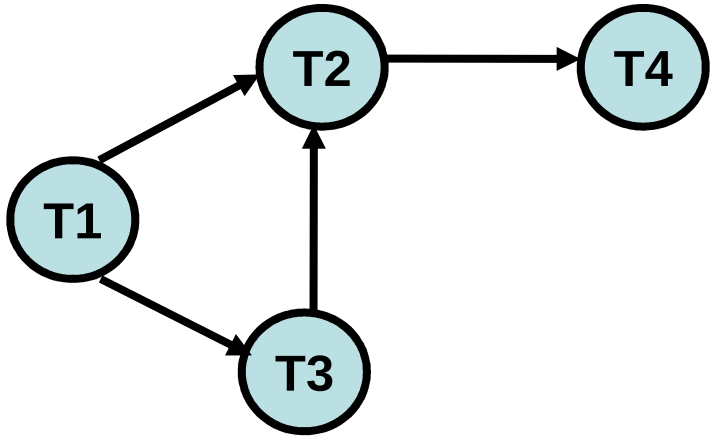
- ✓ Transação que solicitou bloqueio espera por no máximo uma quantidade de tempo especificada.
- ✓ Se esgotou tempo: transação se reverte e reinicia.
- ✓ Esquema fácil de ser implementado.
- ✓ Funciona bem se transações forem curtas e se grandes esperas forem devidas a impasses.
- ✓ Difícil decidir quanto tempo transação deve esperar.

Detecção e recuperação de impasse

- ✓ Detecção:
  - Dado grafo G = (V,E), em que V é um conjunto de vértices e E um conjunto de arestas.
  - Conjunto de vértices: todas as transações do sistema.
  - Cada elemento do conjunto E de arestas é um par ordenado $T_i \rightarrow T_j$. Se $T_i \rightarrow T_k$ está em E ➔ transação T_i está esperando que a transação T_k libere o item de dado que ela precisa.
  - Inserção e remoção de arestas no grafo são feitas de acordo com a solicitação de um item preso ou a liberação de um item preso.
  - Há *deadlock* no sistema se, e somente se, o grafo contiver um ciclo.
  - Cada transação envolvida no ciclo está envolvida no *deadlock*.

Detecção e recuperação de impasse

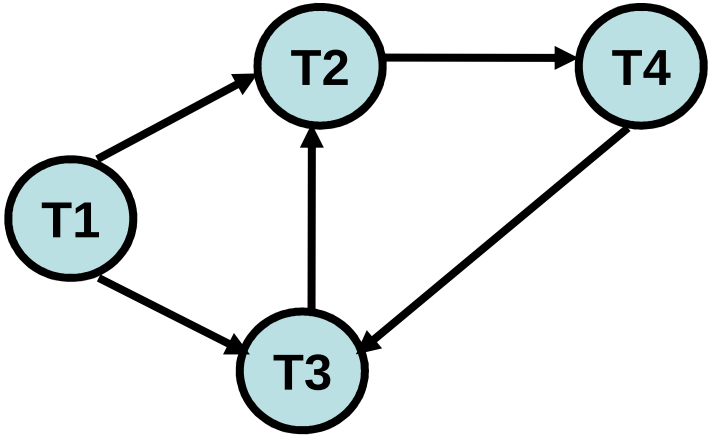
✓ Detecção – Exemplo de grafo de espera sem ciclo:



- T1 está esperando por T2 e T3
- T2 está esperando por T4
- T3 está esperando por T2

Detecção e recuperação de impasse

✓ Detecção – Exemplo de grafo de espera com ciclo:



- T1 está esperando por T2 e T3
- T2 está esperando por T4
- T3 está esperando por T2
- T4 está esperando por T3

Detecção e recuperação de impasse

- ✓ Detecção
  - Quanto invocar algoritmo de detecção:
    - ➢ ver frequência que ocorre um *deadlock*
    - ➢ quantidade de transações afetadas pelo *deadlock*

Detecção e recuperação de impasse

- ✓ Recuperação:
  - Três ações precisam ser tomadas:
    - Selecionar uma vítima (ou vítimas) de acordo com ***custo mínimo***:
      - por quanto tempo a transação executou e quanto tempo continuará execuçao até concluir tarefa;
      - quantos itens de dados a transação usou;
      - quantos itens ainda a transação usará até que se complete;
      - quantas transações serão envolvidas no *rollback*.
    - Rollback: determinar até que ponto ela deve ser revertida:
      - reverter totalmente;
      - o suficiente para quebrar o *deadlock* (exige informações adicionais).
    - Estagnação: garantir que uma transação seja escolhida vítima somente um número finito e pequeno de vezes.

Operações de inserção e exclusão

- ✓ Além do read e write explorados até agora, é preciso verificar:
  - delete(Q): remove o item de dado Q do BD
  - insert(Q): insere um novo item de dado Q no BD e designa um valor inicial para ele.
- ✓ Uma transação T_i que queira operar um read(Q) depois da remoção de Q resulta em erro lógico em T_i.
- ✓ Um transação T_i que quer realizar uma operação de read(Q) antes da inserção de Q, também resultará em um erro lógico em T_i.
- ✓ Também será um erro lógico tentar remover um item de dado inexistente.

Operações de inserção e exclusão

- ✓ Delete entra em conflito com outra instrução quando:
  - uma transação quer ler o item que a outra removeu;
  - uma transação quer escrever o que a outra removeu;
  - uma transação quer remover o que a outra removeu;
  - uma transação quer remover e outra inserir ➔ remoção só pode ser feita se a inserção já ocorreu ou se o dado já existia e a inserção só deve ocorrer se o dado não existe.

- ✓ Se o bloqueio em duas fases for usado, é preciso um bloqueio exclusivo sobre o item de dados antes que ele possa ser removido.

- ✓ Para o protocolo *timestamp* é preciso um controle similar ao usado para o write.

Operações de inserção e exclusão

- ✓ Insert entra em conflito com delete (já visto)
- ✓ Insert entra em conflito com read e write: nenhum *read* ou *write* pode ser realizado sobre um item de dados antes que ele exista
- ✓ Insert é tratado de forma similar a write em termos de concorrência:
  - bloqueio em duas fases: transação necessita de bloqueio exclusivo sobre o item de dados para inserir.
  - ordenação por *timestamp:* se T realizar uma operação insert(Q), valores de T-timestamp(Q) e W-timestamp(Q) são definidos por T.

Níveis de consistência em SQL

- ✓ SQL permite transação que leia registros antes que tenham sido confirmados: útil para transações longas que não precisam de exatidão (exemplo: estatísticas do BD)
- ✓ Níveis possíveis em SQL-92:
  - passíveis de seriação (*default*)
  - leitura repetitiva: registros confirmados e 2 leituras por registro
  - leitura confirmada: somente registros confirmados
  - leitura não confirmada: qualquer registro pode ser lido.

Informações adicionais sobre SGBDs específicos:

- **PostGreSQL**: Silberschatz, A.; Korth, H.F Sudarshan, S. Sistema de Banco de Dados, 5a. edição, Makron Books, 2006 (capítulo 26)
- **Oracle**: Silberschatz, A.; Korth, H.F Sudarshan, S. Sistema de Banco de Dados, 5a. edição, Makron Books, 2006 (capítulo 27)

Bibliografia

- ✓ Silberschatz, A.; Korth, H.F Sudarshan, S. Sistema de Banco de Dados, 5a. edição, Makron Books, 2006 (capítulo 16)
- ✓ Elmasri, R.; Navathe, S. B. Sistemas de Banco de Dados. São Paulo: Addison Wesley, 2005.

Exercícios

- ✓ Silberschatz, A.; Korth, H.F Sudarshan, S. Sistema de Banco de Dados, 5a. edição, Makron Books, 2006 (capítulo 16) – Exercícios 1 a 5, 11, 12, 16, 18, 19, 22 a 25
- ✓ Elmasri, R.; Navathe, S. B. Sistemas de Banco de Dados. São Paulo: Addison Wesley, 2005. Exercícios 1, 3, 4, 7, 12, 14, 15, 20,

ACH2025

Laboratório de Bases de Dados

Aula 17

Controle de Concorrência

Professora:

- **Fátima L. S. Nunes**

USP

SISTEMAS DE INFORMAÇÃO

EACH