PUCRS
Pontifícia Universidade Católica
do Rio Grande do Sul

ESCOLA
POLITÉCNICA

Controle de Concorrência

98902-02 - Banco de Dados II

Prof. Msc. Eduardo Arruda
eduardo.arruda@pucrs.br

Propriedades ACID

A Atomicidade

C Consistência

I Independência

D Durabilidade

Isolamento

- A execução de uma transação Tx deve funcionar como se Tx executasse de forma isolada
  - Tx não deve sofrer interferências de outras transações executando concorrentemente
  - Resultados intermediários das transações devem ser escondidos de outras transações executadas concorrentemente
  - Responsabilidade do subsistema de controle de concorrência (scheduler) do SGBD
  - Diversos problemas de interferência de transações

Problema 1 – Atualização Perdida (*Lost Update*)

- Resevas de cia aérea armazenado registros de Voo da cia
- Cada registro contem o número de poltronas reservada para o voo.
- Figura ao lado representa : T1 transfere N reservas de um voo, cujo numero de assentos reservados está armazenado em um item de dados chamado X, para um outro voo, cujo número de de assentos reservados está armazenado em um item de dados chamado Y.

X	Y
READ(X)	
X=X-N	
	READ(X)
	X=X+M
WRITE(X)	
READ(Y)	
	WRITE(X)
Y=Y+N	
WRITE(Y)	

Qual o problema???

- Se duas transações acessarem os mesmos itens com operações intercaladas.
- Suponha T1 e T2, sejam submetidas aproximadamente ao mesmo tempo.
- Qual o valor final de X?
- T2 – Lerá o valor de x antes de T1 mudá-lo no banco, portanto o valor atualizado resultando de T1 será perdido.
- Testem com os seguintes valores
- X=80
- N=5
- M=4
- O resultado deveria ser 79 porém resulta 84 pois perdeu-se a remoção dos 5 assentos.

X	Y
READ(X)	
X=X-N	
	READ(X)
	X=X+M
WRITE(X)	
READ(Y)	
	WRITE(X)
Y=Y+N	
WRITE(Y)	

item X tem um valor incorreto porque a atualização feita por T1 foi		
"perdida" (sobrescrita)		

Atualização Perdida

- Uma transação T1 grava em um dado atualizado por uma transação T2

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = X + 20	
write(Y)	

a atualização de X por T1 foi perdida!

Problema 2 – Leitura Suja (*Dirty Read*)

- T1 atualiza um item de bd, e a seguir ocorre um problema que a leva a ser abortada
- Logo em seguida a T2 acessa estes dados antes de T1 fazer rollback!? E agora?!!
- T2 nao pode ser gravado em BD por que T1 falhou, o BD deve tratar esta inconsistência
- Rollback em cascata.
- E se T1 já tiver feito commit?

T1	T2
X	Y
READ(X)	
X=X-N	
WRITE(X)	
	READ(X)
	X=X+M
	WRITE(X)
READ(Y)	

Problema

Rollback

Leitura Suja

- T1 atualiza um dado X, outras transações posteriormente lêem X, e depois T1 falha

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
abort( )	

T_2 leu um valor de X que não será mais válido!

Problema 3: Leituras não repetíveis (*Unrepeatable Reads*)

- Quando uma transação lê mais de uma vez um mesmo dado e obtém valores diferentes
- Imagine um sistema de reservas de passagens onde o sistema verifica se existem lugares disponíveis, mostra os lugares para o usuário e este confirma a compra, momento onde a reserva dos lugares é efetuada. Porém se entre a primeira leitura e a reserva ocorrer outra venda o sistema não poderá fazer esta reserva senão pode fazer reservas a mais.

Problema 4 – Sumário Incorreto (Summary Problem)

- T3(Y) Está calculando o número total de reservas em todos os voos
- Enquanto isso... T1 começa a a ser executada
- Se acontecer a intercalação de operações mostrado ao lado
- O resultado de T3 não contabilizará N, pois T3 leu o valor de X depois que os N assentos foram subtraídos, mas lerá o valor Y antes que esses N assentos tenham sido somados a ele.

X	Y
	sum=0
	READ(A)
	SUM=SUM+A
	.
	.
	.
READ(X)	
X=X+N;	
WRITE(X)	
	READ(A)
	SUM=SUM+X
	READ(Y)
	SUM=SUM+Y
READ(Y)	
Y=Y+N	
WRITE(Y)	

Total de reservas

Levando valores de y

Controle de Concorrência

- Garantia de isolamento de Transações
  - Solução 1 : Uma transação executa por vez
  - Solução 2: Execução concorrente de transações

Execução serial

T1	T2
read(X)	
X = X – 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

Execução concorrente ou não-serial

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

Escalonador

- Responsável pela definição de escalonamentos não-seriais de transações
- “Um escalonamento E define uma ordem de execução das operações de várias transações, sendo que a ordem das operações de uma transação Tx em E aparece na mesma ordem na qual elas ocorrem isoladamente em Tx”

Escalonador

- Deve evitar escalonamentos inválidos
  - Atualização perdida
  - Leitura suja
  - Sumário Incorreto
- Exige análise de operações conflitantes

Escalas ou histórias

- Representação seqüencial da execução entrelaçada de um conjunto de transações concorrentes
  - operações consideradas
    - read (r), write (w), commit (c), abort (a)
- Uma escala para um conjunto de transações deve consistir de todas as instruções de todas as transações
- Deve preservar a ordem na qual as instruções aparecem em cada transação individual

Exemplos de Escalas

- Suponha que T1 transfere R$50 da conta A para a conta B, e T2 transfere 10% do saldo de A para B.
- A escala seguinte é SERIAL: T1 é seguida por T2

Escala 1:

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

inicial:

A = 1000

B = 2000

soma = 3000

depois de T1:

A = 950

B = 2050

depois de T2:

A = 855

B = 2145

soma = 3000

Exemplos de Escalas

- Escala 2: SERIAL com T2 antes de T1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

inicial:

A = 1000

B = 2000

soma = 3000

depois de T2:

A = 900

B = 2100

depois de T1:

A = 850

B = 2150

soma = 3000 (PRESERVADA!)

Exemplos de Escalas

- Escala 3: não é serial, mas é equivalente à Escala 1

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

A = 1000

B = 2000

A = 950

A = 855

B = 2050

B = 2145

855 + 2145 = 3000

SOMA PRESERVADA!

Exemplos de Escalas

- Escala 4: não preserva a soma dos saldos de A e B

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

A = 1000

B = 2000

A = 950

A = 900

2050

B = 2100

final: A = 950, B = 2100

Operações conflitantes

- Podem gerar um estado inconsistente da base de dados
- Operações conflitantes
  - operações que pertencem a transações diferentes
  - transações acessam o mesmo dado
  - pelo menos uma das operações é write

		T1	
		read(X)	write(X)
T2	read(X)		√
	write(X)	√	√

Exemplo de Escala

- Escalonamento

  r1(X) r2(X) w1(X) r1(Y) w2(X) w1(Y)

- Operações conflitantes ?
  - ? e ?
  - ? e ?
  - ? e ?

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = X + 30	
write(Y)...	...

Escalonamentos X recuperação

- Recuperação de falhas
  - O processo de recuperação de falhas é dependente da estratégia de escalonamento
  - Por isso, é importante caracterizar os tipos de escalonamentos para os quais é possível a recuperação

Escalonamentos X Recuperação

- Scheduler deve cooperar com o Recovery!
- Categorias de escalonamentos
  - Recuperáveis X não-recuperáveis
  - Evitam aborto em cascata X permitem aborto em cascata X
  - Estritos X não-estritos

Escalonamento Recuperável

- Um escalonamento E é recuperável se nenhuma transação T2 for concluída até que todas as transações que gravaram dados lidos por T2 tenham sido concluídas

Escalonamento não-recuperável

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
	commit( )
abort( )	

Gravou (X)

Finalizado antes

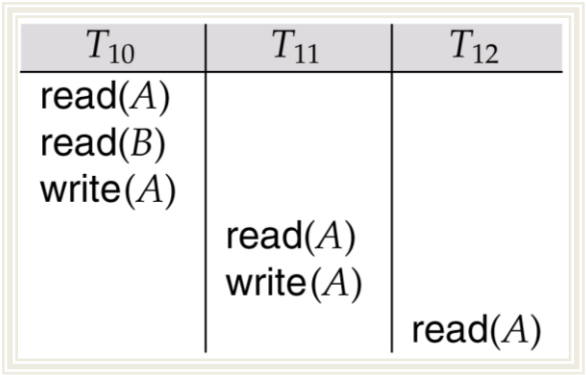
Escalonamento recuperável

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
commit( )	
	commit( )

Leu (x)da T1

Aborto em cascata

- *Aborto* em cascata – uma falha em uma única transação pode levar a uma série de *aborts*. Considere a escala a seguir, onde nenhuma das transações foi confirmada (ou seja, a escala é recuperável)



T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- Se T10 falhar, T11 deverá ser abortada e consequentemente T12
- Isso pode levar ao cancelamento de uma quantidade significativa de trabalho

Escalonamento sem Aborto em Cascata

- Um escalonamento E é recuperável e evita aborto em cascata se uma transação T2 só puder ler dados que tenham sido atualizados por transações que já concluíram

escalonamento recuperável **com aborto em cascata**

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
abort( )	. . .

escalonamento recuperável **sem aborto em cascata**

T1	T2
read(X)	
X = X – 20	
write(X)	
commit( )	
	read(X)
	X = X + 10
	write(X)
	. . .

Escalonamento Estrito

- Um escalonamento E é recuperável, evita aborto em cascata e é estrito se uma transação T2 só puder ler ou atualizar um dado X depois que todas as transações que atualizaram X tenham sido concluídas

escalonamento recuperável sem aborto em cascata e **não-estrito**

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(Y)
	X = Y + 10
	write(X)
	commit( )
commit( )	

escalonamento recuperável sem aborto em cascata e **estrito**

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(Y)
	X = Y + 10
commit( )	
	write(X)
	commit( )

Exercício

H_E = w1(x) w1(y) r2(u) w2(x) w1(z) c1 r2(y) w2(y) c2

- O escalonamento acima é
  - recuperável?
  - evita aborto em cascata?
  - estrito?

Teoria da Serializabilidade

- A execução de um conjunto de transações é dita serializável se, e somente se, ela produz os mesmos resultados e os mesmos efeitos no BD que alguma das possíveis execuções seriais do mesmo conjunto de Transações
- Já que a execução de uma Transação consistente sempre leva o BD de um estado consistente a outro estado consistente, então execuções seriais (de Ts consistentes) também são corretas
- Se execuções seriais são sempre corretas e qualquer execução serializável é sempre equivalente a alguma execução serial, então execuções serializáveis são também consideradas corretas

Teoria da Serializabilidade

- Garantia de escalonamentos não-seriais válidos
- Premissa
  - "um escalonamento não-serial de um conjunto de transações deve produzir resultado equivalente a alguma execução serial destas transações"

entrada:
X = 50
Y = 40

execução serial

saída:
X = 40
Y = 60

T1	T2
read(X)	
X = X – 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

entrada:
X = 50
Y = 40

execução não-serial ***serializável***

saída:
X = 40
Y = 60

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

Verificação de Serializabilidade

- Equivalência de Conflito

“ dois escalonamentos não-seriais são considerados conflito equivalentes se a ordem de quaisquer duas operações conflitantes for a mesma em ambos os escalonamentos”

Equivalência de Conflito - Exemplo

escalonamento serial *E*

T1	T2
read(X)	
X = X – 20	
write(X)	
read(Y)	
Y = Y + 20	
write(Y)	
	read(X)
	X = X + 10
	write(X)

escalonamento não-serial *E1*

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	

escalonamento não-serial *E2*

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = Y + 20	
write(Y)	

- *E1* equivale em conflito a *E*
- *E2* não equivale em conflito a nenhum escalonamento serial para T1 e T2
- *E1* é serializável e *E2* não é serializável

Casos a considerar conflito entre ti..tj

- Quatro casos a considerar:
  - read(q), read(q): seqüência de execução não importa, pois o mesmo valor é lido
  - read(q), write(q): seqüência de execução importa: lê antes, escreve depois ou escreve antes e lê depois
  - write(q), read(q): seqüência de execução importa: lê antes, escreve depois ou escreve antes e lê depois
  - write(q), write(q): importa, pois o valor final gerado depende de quem executar por último (e isso afeta quem for ler o valor depois)

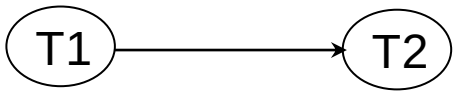
Verificação de Equivalência em Conflito

- Construção de um grafo direcionado de precedência
  - nodos são IDs de transações
  - arestas rotuladas são definidas entre duas transações T1 e T2 se existirem operações em conflito entre elas
    - direção indica a ordem de precedência da operação
- Um grafo com ciclos indica um escalonamento não-serializável em conflito!

Grafo de Precedência

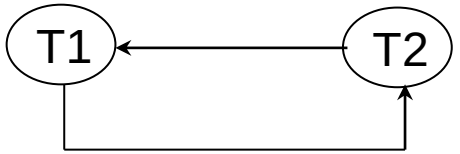
escalonamento serializável *E1*

T1	T2
read(X)	
X = X – 20	
write(X)	
	read(X)
	X = X + 10
	write(X)
read(Y)	
Y = Y + 20	
write(Y)	



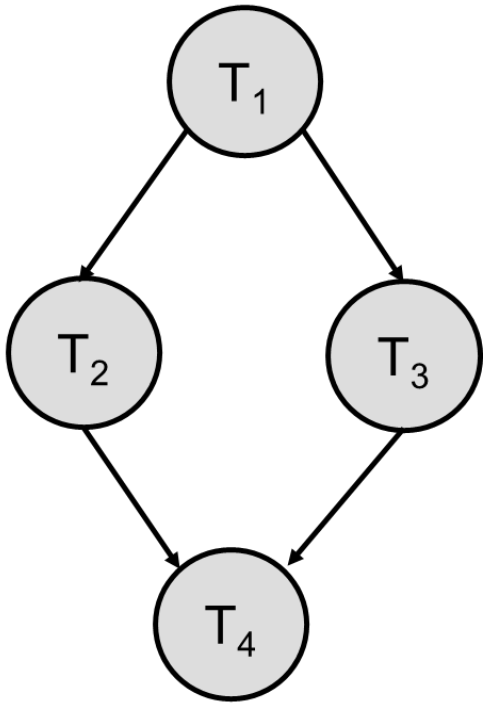
escalonamento não-serializável *E2*

T1	T2
read(X)	
X = X – 20	
	read(X)
	X = X + 10
write(X)	
read(Y)	
	write(X)
Y = Y + 20	
write(Y)	



Grafo de Precedência – E3

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

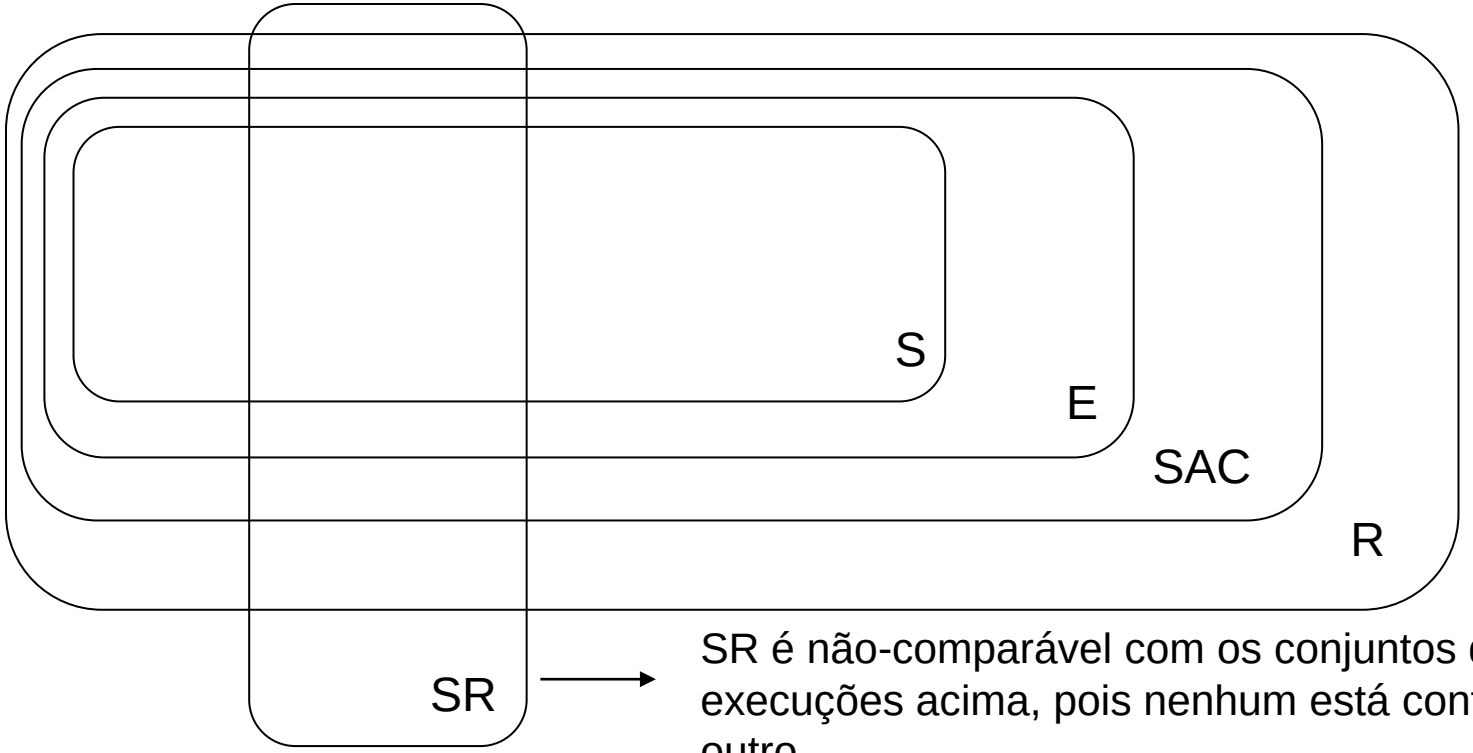


Teste de Serialização de Conflito

- Uma escala é serializável em condição de conflito se e somente se o seu grafo de precedência é ACÍCLICO
- Ex: a escala E1 é acíclica → serializável
- Ex: a escala E2 é cíclica → não serializável
- Ex: a escala E3 é acíclica → serializável
- Há algoritmos de detecção de ciclos com esforço computacional na ordem de n2, onde n é o nro de vértices no grafo. Algoritmos melhores possuem ordem n + ‘e’, onde ‘e‘é o nro de arestas

Relação entre Escalonamentos

SR = escalonamento serializável
R = escalonamento recuperável
SAC = escalonamento sem aborto em cascata
E = escalonamento estrito
S = escalonamento serial



SR é não-comparável com os conjuntos de execuções acima, pois nenhum está contido no outro

Verificação de Serializabilidade

- Na prática é difícil testar a serializabilidade de um escalonamento
  - Exige que se tenha um conjunto fechado de transações para fins de verificação
  - Novas transações estão sendo constantemente submetidas ao SGBD para execução
  - O entrelaçamento de operações de transações concorrentes é determinado, em geral, pelo escalonamento do Sistema Operacional
  - Logo, normalmente a serializabilidade é garantida através de protocolos de controle de concorrência, que não precisam testar os escalonamentos, mas garantem que os escalonamentos gerados são serializáveis

Técnicas de Controle de Concorrência

- Pessimistas
  - supõem que sempre ocorre interferência entre transações e garantem a serializabilidade enquanto a transação está ativa
  - técnica
    - bloqueio (*locking*)
- Otimistas
  - supõem que quase nunca ocorre interferência entre transações e verificam a serializabilidade somente ao final de uma transação
  - técnica
    - Validação (já vista)
    - *timestamp*

Exercícios

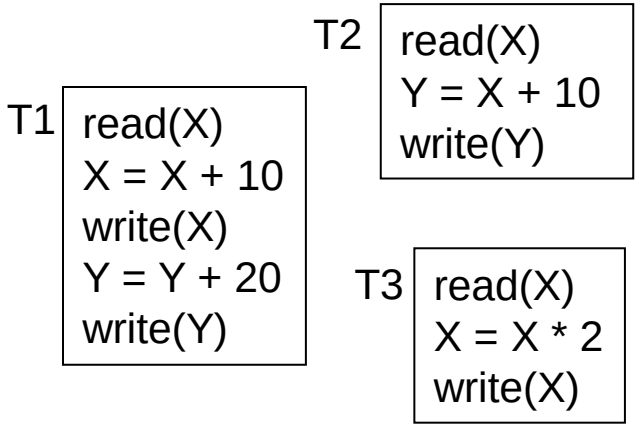
1. Dadas as transações abaixo, associe corretamente a história com o tipo de escalonamento: Serial (SR), Não-Recuperável (NR), Recuperável (R), Sem Aborto em Cascata (SAC), Estrito (E) e Serializável (SL).

T1 = w(x) w(y) w(z) c1 T2 = r(u) w(x) r(y) w(y) c2

HE1 = w1(x) w1(y) r2(u) w2(x) r2(y) w2(y) c2 w1(z) c1 ( )
HE2 = w1(x) w1(y) w1(z) c1 r2(u) w2(x) r2(y) w2(y) c2 ( )
HE3 = w1(x) w1(y) r2(u) w2(x) w1(z) c1 r2(y) w2(y) c2 ( )
HE4 = w1(x) w1(y) r2(u) w1(z) c1 w2(x) r2(y) w2(y) c2 ( )
HE5 = w1(x) r2(u) w2(x) r2(y) w2(y) w1(y) w1(z) c1 c2 ( )

Exercícios

2. Dadas as transações ao lado, dê um exemplo de uma história/escalonamento:
   a) não-serializável
   b) serializável e não-recuperável
   c) sem aborto em cascata



Exercícios

3. Construa o grafo de precedência e diga se os escalonamentos são serializáveis. Em caso positivo, qual seria a execução serial equivalente?

T1 = r(x) w(x) r(y) w(y) ;
T2 = r(z) r(y) w(z) r(x) w(x);
T3 = r(y) r(z) w(y) w(z);
E1 = r2(z) r2(y) w2(y) r3(y) r3(z) r1(x) w1(x)
w3(y) w3(z) r2(x) r1(y) w1(y) w2(x)
E2 = r3(y) r3(z) r1(x) w1(x) w3(y) w3(z) r2(z)
r1(y) w1(y) r2(y) w2(y) r2(x) w2(x)

Exercícios

4. Qual dos planos é serializável em conflito? Qual o plano serial equivalente?

a) r1(X);r3(X);w1(X);r2(X);w3(X)
b) r1(X);r3(X);w3(X):W1(X);r2(X)
c) r3(X);r2(X);w3(X);r1(X);w1(X)
d) r3(X);r2(X);r1(X);w3(X);w1(X)

Exercícios

5. Fazer os grafos de serialidade e dizer se é serializável ou não e qual seria o plano serial equivalente:

T1 = r1(X);r1(Z);w1(X)
T2 = r2(Z);r2(Y);w2(Z);w2(Y)
T3 = r3(X);r3(Y);w3(Y)
E1 = r1(X) r2(Z) r1(Z) r3(X) r3(Y) w1(X) w3(Y) r2(Y) w2(Z) w2(Y)
E2 = r1(X) r2(Z) r3(X) r1(Z) r2(Y) r3(Y) w1(X) w2(Z) w3(Y) w2(Y)

Exercícios

6. Para as mesmas transações do exercício 5 e para os seguintes escalonamentos, determinar se os planos são restritos, livres de cascatas, recuperáveis ou não-recuperáveis

a) E3 = r1(X) r2(Z) r1(Z) r3(X) r3(Y) w1(X) c1 w3(Y) c3 r2(Y) w2(Z) w2(Y) c2;
b) E4 = r1(X) r2(Z) r1(Z) r3(X) r3(Y) w1(X) w3(Y) r2(Y) w2(Z) w2(Y) c1 c2 c3;
c) E5 = r1(X) r2(Z) r3(X) r1(Z) r2(Y) r3(Y) w1(X) c1 w2(Z) w3(Y) w2(Y) c3 c2;

Técnicas de Controle de Concorrência

- Pessimistas → Baseadas em bloqueios
- Otimistas → Baseadas em *timestamps*

Técnicas Baseadas em Bloqueio

- Técnicas mais utilizadas por SGBDs
- Princípio de funcionamento
  - controle de operações read(X) e write(X) e postergação (através de bloqueio) de algumas dessas operações de modo a evitar conflito
- Todo dado possui um *status* de bloqueio
  - **liberado** (*Unlocked* - U)
  - com **bloqueio compartilhado** (*Shared lock* - S)
  - com **bloqueio exclusivo** (*eXclusive lock* - X)

Modos de Bloqueio

- Bloqueio Compartilhado (S)
  - solicitado por uma transação que deseja realizar leitura de um dado D
    - várias transações podem manter esse bloqueio sobre D
- Bloqueio Exclusivo (X)
  - solicitado por uma transação que deseja realizar leitura + atualização de um dado D
    - uma única transação pode manter esse bloqueio sobre D
- Matriz de compatibilidade de bloqueios

	S	X
S	verdadeiro	falso
X	falso	falso

- Informações de bloqueio são mantidas no Dicionário de Dados
  - <ID-dado, status-bloqueio, ID-transação>

Operações de bloqueio no escalonamento

- O Escalonador gerencia bloqueios através da invocação automática de operações de bloqueio conforme a operação que a transação deseja realizar em um dado
- Operações
  - ls(D): solicitação de bloqueio compartilhado sobre D
  - lx(D): solicitação de bloqueio exclusivo sobre D
  - u(D): libera o bloqueio sobre D

Exemplo de História com Bloqueios

H = ls1(Y) r1(Y) u1(Y) ls2(X) ls2(Y)
r2(X) r2(Y) u2(X) lx2(Y) w2(Y)
u2(Y) c2 lx1(X) r1(X) w1(X)
u1(X) c1

T1	T2
lock-S(Y)	
read(Y)	
unlock(Y)	
	lock-S(X)
	lock-S(Y)
	read(X)
	read(Y)
	unlock(X)
	lock-X(Y)
	write(Y)
	unlock(Y)
	commit( )
lock-X(X)	
read(X)	
write(X)	
unlock(X)	
commit( )	

Solicitação de bloqueio compartilhado

```
lock-S(D, Tx)                                   ──→ status de bloqueio de D
início
        se lock(D) = 'U' então
        início
                insere Tx na lista-READ(D);     ──→ lista de transações
                lock(D) ← 'S';                      com bloqueio
        fim                                         compartilhado sobre D
        senão se lock(D) = 'S' E fila-WAIT(D) é vazia então
                insere Tx na lista-READ(D)
              senão  /*  lock(D) = 'X' */
                insere (Tx, 'S') na fila-WAIT(D);
                                                ──→ fila de transações aguardando a
fim                                                 liberação de um bloqueio
                                                    conflitante sobre D
```

Solicitação de bloqueio exclusivo

```
lock-X(D, Tx)                                   ──→ status de bloqueio de D
início
        se lock(D) = 'U' então
        início
                insere Tx na variável WRITE(D);  ──→ Transação com bloqueio exclusivo sobre D
                lock(D) ← 'X';
        fim
        senão insere (Tx, 'X') na fila-WAIT(D); /*  lock(D) = 'S' ou 'X' */
fim
```

fila de transações aguardando a liberação de um bloqueio conflitante sobre D

Solicitação de desbloqueio

```
Unlock (D, Tx)
início
        se lock(D) = 'S' então
        início
                retira Tx da lista-READ(D);
                se lista-READ(D) está vazia então
                  inicio
                     lock(D) ← 'U';
                     processa fila fila-WAIT(D);
                   fim
        fim
        senão /*  lock(D) = 'X' */
        inicio
                limpa variável WRITE(D);
                lock(D) ← 'U';
                processa fila-WAIT(D);
        fim
fim
```

status de bloqueio de D

lista de transações com bloqueio compartilhado sobre D

fila de transações aguardando a liberação de um bloqueio conflitante sobre D

Uso de Bloqueios “S” e “X”

- Não garantem escalonamentos serializáveis
- Exemplo

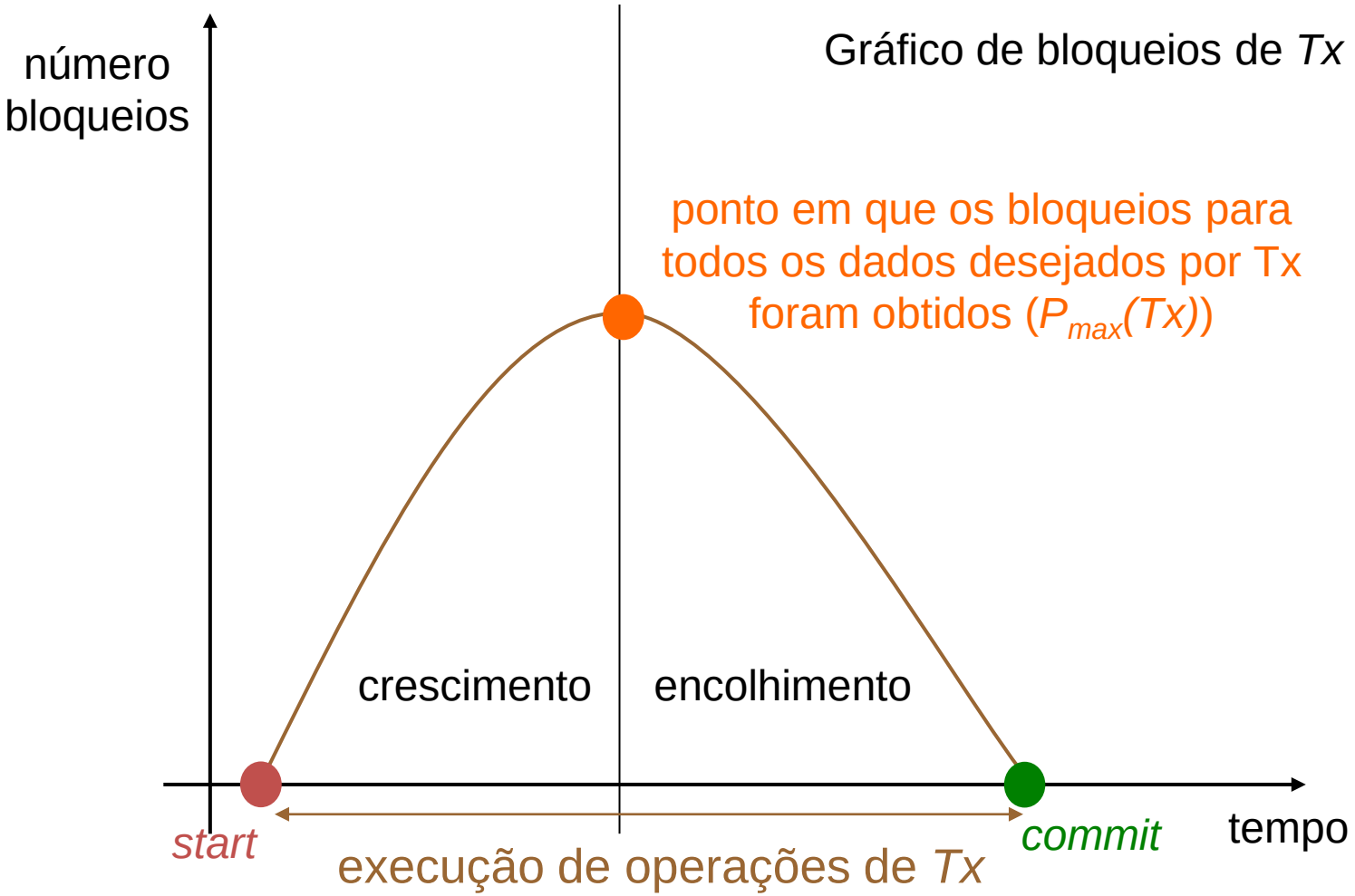
$H_{N\text{-}SR}$ = ls1(Y) r1(Y) u1(Y) ls2(X) r2(X) u2(X) lx2(Y) r2(Y)
w2(Y) u2(Y) lx1(X) r1(X) w1(X) u1(X) c2 c1

- Necessita-se de uma técnica mais rigorosa de bloqueio para garantir a serializabilidade
  - técnica mais utilizada
    - bloqueio de duas fases (*two-phase locking* – 2PL)

Bloqueio de 2 Fases – 2PL

- Premissa
  - “para toda transação Tx, todas as operações de bloqueio de dados feitas por Tx precedem a primeira operação de desbloqueio feita por Tx”
- Protocolo de duas fases
  - Fase de expansão ou crescimento
    - Tx pode obter bloqueios, mas não pode liberar nenhum bloqueio
  - Fase de retrocesso ou encolhimento
    - Tx pode liberar bloqueios, mas não pode obter nenhum bloqueio

Escalonador 2PL – Funcionamento



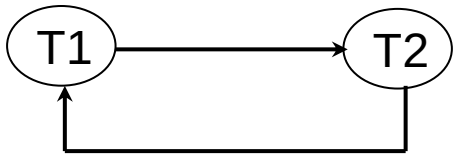
Escalonador 2PL - Exemplo

T1: r(Y) w(Y) w(Z)
T2: r(X) r(Y) w(Y) r(Z) w(Z)

não é 2PL!

Contra-Exemplo

HN-2PL = lx1(Y) r1(Y) ls2(X) r2(X) u2(X) w1(Y) u1(Y) lx2(Y)
r2(Y) w2(Y) u2(Y) lx2(Z) r2(Z) w2(Z) c2 lx1(Z) w1(Z)
u1(Z) c1



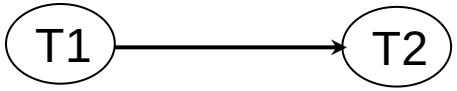
não garantiu SR!

$P_{max}(T1)$

Exemplo

H_{2PL} = ls2(X) r2(X) lx1(Y) r1(Y) lx1(Z) w1(Y) u1(Y) lx2(Y)
r2(Y) w1(Z) u1(Z) c1 w2(Y) lx2(Z) u2(X) u2(Y) w2(Z)
u2(Z) c2

$P_{max}(T2)$



é SR!

Escalonador 2PL - Crítica

- Vantagem
  - técnica que sempre garante escalonamentos SR sem a necessidade de se construir um grafo de dependência para teste
    - se Tx alcança Pmax, Tx não sofre interferência de outra transação Ty, pois se Ty deseja um dado de Tx em uma operação que poderia gerar conflito com Tx, Ty tem que esperar (evita ciclo Ty $\rightarrow$ Tx)
    - depois que Tx liberar os seus dados, não precisará mais deles, ou seja, Tx não interferirá nas operações feitas futuramente nestes dados por Ty (evita ciclo Tx $\rightarrow$ Ty)

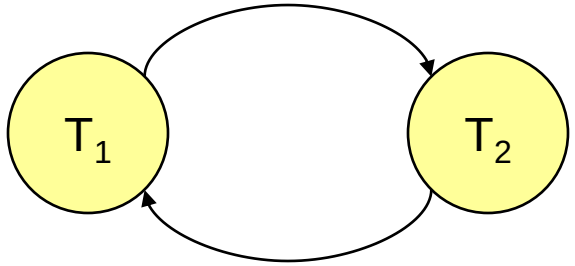
Escalonador 2PL - Crítica

- Desvantagens
  - limita a concorrência
    - um dado pode permanecer bloqueado por Tx muito tempo até que Tx adquira bloqueios em todos os outros dados que deseja
  - 2PL básico (técnica apresentada anteriormente) não garante escalonamentos livres de deadlock
    - Tx espera pela liberação de um dado bloqueado por Ty de forma conflitante e vice-versa
  - Não garante escalonamentos recuperáveis (por isso mesmo, nem sem aborto em cascata).

Deadlock (Impasse) de Transações

- Uma transação T1 espera por um objeto bloqueado por uma transação T2 enquanto que T2 espera por um objeto bloqueado por T1

T_1	T_2
lock-X(y)	
read(y)	
write(y)	
	lock-S(x)
lock-X(x)	**read(x)**
	lock-X(y)



Grafo Wait-For

Deadlock (Impasse) de Transações

- Ocorrência de *deadlock*
  - Tx mantém um dado D1 bloqueado e na Fila-WAIT(D1) está Ty
  - Ty mantém um dado D2 bloqueado e na Fila-WAIT(D2) está Tx
- Se há *deadlock*, seleciona-se uma transação vítima Tx através de um ou mais critérios
  - quanto tempo Tx está em processamento
  - quantos itens de dado Tx já leu/escreveu
  - quantos itens de dado Tx ainda precisa ler/escrever
  - quantas outras transações serão afetadas pelo abort(Tx)
- Pode ser descoberto através de um grafo de espera de transações
  - se o grafo é cíclico existe *deadlock*!

Tratamento de *Deadlocks*

- Protocolos de Prevenção
  - Abordagens pessimistas:
    - *deadlocks* ocorrem com frequência e impõem um *overhead* no processamento de transações
    - controles adicionais para evitar *deadlock*
  - Tipos de protocolos pessimistas
    - técnica de bloqueio 2PL conservador
    - técnicas baseadas na ordenação das transações (*wait-die* e *wound-wait*)
      - A transação mais recente é abortada e reinicia com a mesma ordem
    - técnica de espera-cautelosa (*cautious-waiting*)
    - uso de timeout: se tempo de espera de Tx > timeout → abort(Tx)

Protocolo de Prevenção de *Deadlock*

- *Timestamp*
  - Ordem total entre as transações
    - Se T1 começa antes de T2 então TS(T1) < TS(T2)
- 2 protocolos de prevenção de *deadlocks*
  - Espera segundo ordem do *timestamp*
  - Espera sempre na mesma ordem previne *deadlock*
  - Idéia: favorecer a transação mais antiga
  - Protocolos: *Wait-die* e *Wound-wait*

Wait-Die

- Situação: T tenta bloquear um objeto já bloqueado por T’

  T solicita um *lock* que está com T'
  Se TS(T) < TS(T')
  Então T espera
  Caso contrário abortar T

- Transações mais velhas podem esperar
- Transações mais jovens são abortadas e re-inicializadas com o mesmo *timestamp*

Wound-Wait

- Situação: T tenta bloquear um objeto já bloqueado por T'

  T solicita um *lock* que está com T'
  Se TS(T) < TS(T')
  Então aborta T'
  Caso contrário T espera

- Transações mais velhas liquidam as mais jovens que são re-inicializadas com o mesmo *timestamp*
- Transações mais jovens podem esperar

Tratamento de *Deadlock*

- Protocolos sem Prevenção
  - abordagens otimistas
    - *deadlocks* não ocorrem com freqüência e são tratados quando ocorrem
  - Mantém-se um grafo de espera de transações e se há *deadlock*, seleciona-se uma transação vítima Tx através de um ou mais critérios
    - quanto tempo Tx está em processamento
    - quantos itens de dado Tx já leu/escreveu
    - quantos itens de dado Tx ainda precisa ler/escrever
    - quantas outras transações serão afetadas pelo abort(Tx)

Exercícios

1. Apresente um início de escalonamento 2PL básico que recaia em uma situação de *deadlock*
2. Apresente um escalonamento 2PL básico que não seja recuperável
    - lembrete: um escalonamento é recuperável se Tx nunca executa commit antes de Ty, caso Tx tenha lido dados atualizados por Ty

Resposta 1

T1	T2
lock-X(Y)	
read(Y)	
write(Y)	
	lock-X(X)
	read(X)
	write(x)
lock-S(X)	
	lock-S(Y)

T1 espera T2 liberar lock-X(X)

T2 espera T1 liberar lock-X(Y)

Deadlock!!!!

Resposta 2

T1	T2
lock-X(Y)	
read(Y)	
write(Y)	
lock-X(z)	
Read(z)	
unlock(Y)	
	lock-X(Y)
	read(Y)
	write(Y)
	unlock(Y)
	commit
write(z)	
abort	

T2 lê dado gravado por T1

T2 realiza commit antes de T1 realizar commit

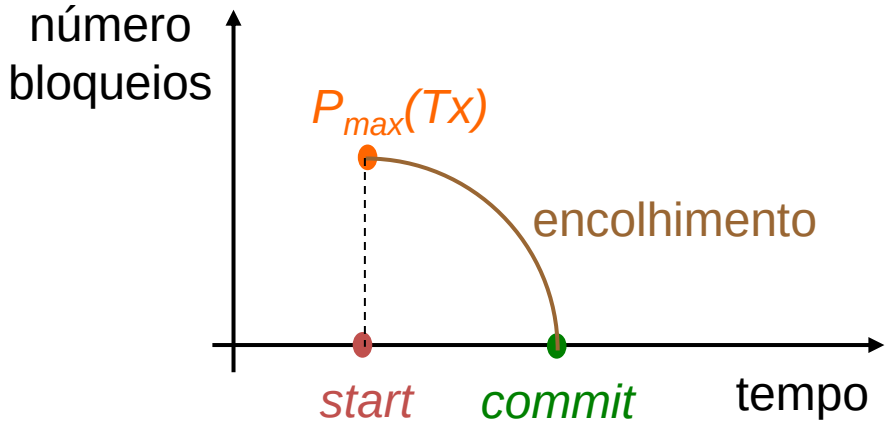
Este escalonamento não é recuperável apesar de respeitar protocolo 2PL

Variações de 2PL

- Escalonador 2PL Básico: já apresentado
- Escalonador 2PL Conservador ou Estático
- Escalonador 2PL Estrito
- Escalonador 2PL Rigoroso

Escalonador 2PL Conservador

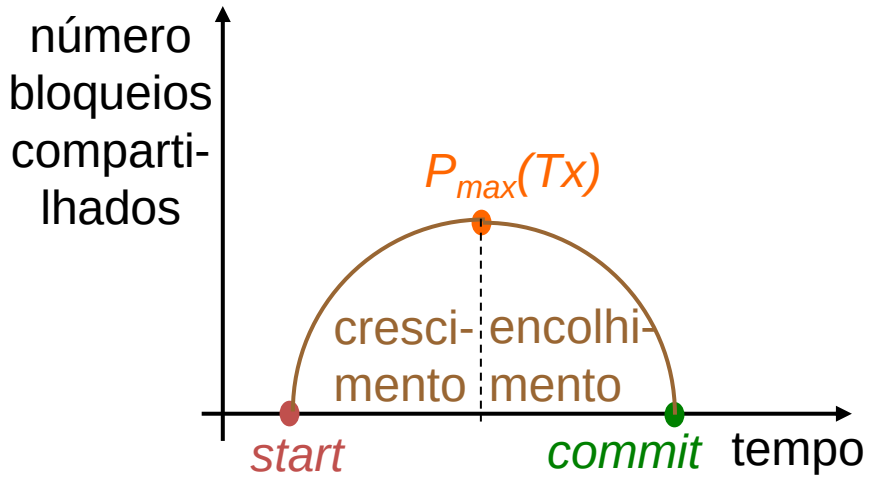
- Tx deve bloquear todos os dados que deseja antes de iniciar qualquer operação
  - Caso não seja possível bloquear todos os dados, nenhum bloqueio é feito e Tx entra em espera para tentar novamente

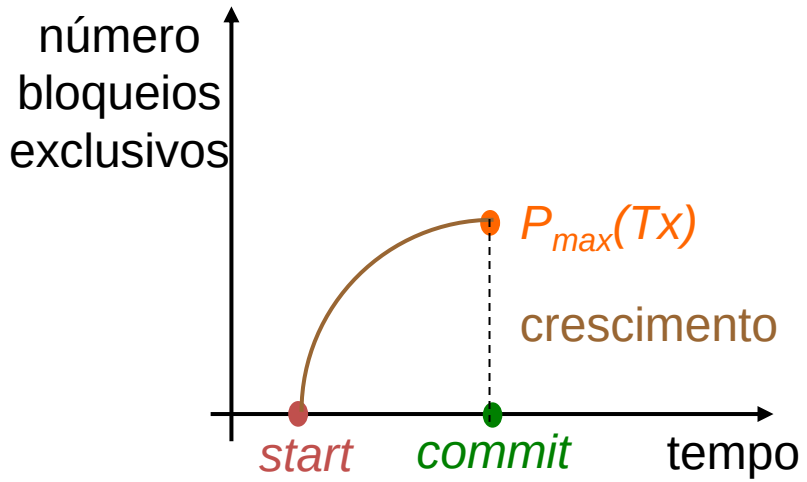


  - Vantagem
    - uma vez adquiridos todos os seus bloqueios, Tx não entra em *deadlock* durante a sua execução
  - Desvantagem
    - espera pela aquisição de todos os bloqueios!

Escalonador 2PL Estrito
(muito usado por SGBDs)

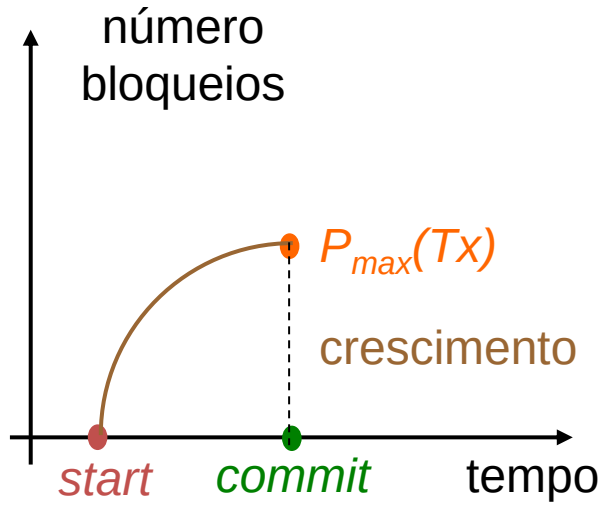
- Tx só libera seus bloqueios exclusivos após executar commit ou abort
- Vantagem: garante escalonamentos estritos
- Desvantagem: não está livre de *deadlocks*





Escalonador 2PL (estrito) Rigoroso

- Tx só libera seus bloqueios (exclusivos e compartilhados) após executar commit ou abort
- Vantagem
  - menos overhead para Tx
    - Tx libera tudo apenas no final
    - Não identifica Read-Set e Write-Set
- Desvantagem
  - limita mais a concorrência



Comparativo

- Escalonador 2PL Conservador
  - Bloqueia tudo entes de iniciar
  - Quando a transação inicia ela está na sua fase de encolhimento
- Escalonador 2PL Estrito
  - Só desbloqueia itens com bloqueio exclusivo depois de terminar
  - Garante escalonamento estrito, mas não evita *deadlocks*
- Escalonador 2PL Rigoroso
  - Só desbloqueia qualquer item depois de terminar
  - Limita a concorrência

Exercícios

1. Apresente exemplos de escalonamentos 2PL conservador, 2PL estrito e 2PL rigoroso para as seguintes transações:

T1: r(Y) w(Y) w(Z)
T2: r(X) r(T) w(T)
T3: r(Z) w(Z)
T4: r(X) w(X)

2. Apresente uma situação de *deadlock* em um escalonamento 2PL estrito

Técnicas baseadas em *Timestamp*

- Técnica pela qual toda transação Tx possui uma marca de *timestamp* (TS(Tx))
- Princípio do funcionamento (básico)
  - "no acesso a um item de dado D por operações conflitantes, a ordem desse acesso deve ser equivalente à ordem de TS das transações envolvidas"
  - Garante escalonamentos serializáveis através da ordenação de operações conflitantes de acordo com os TSs das transações envolvidas
  - Cada item de dado X possui um registro TS (R-TS(X))

<ID-dado, TS-Read, TS-Write>

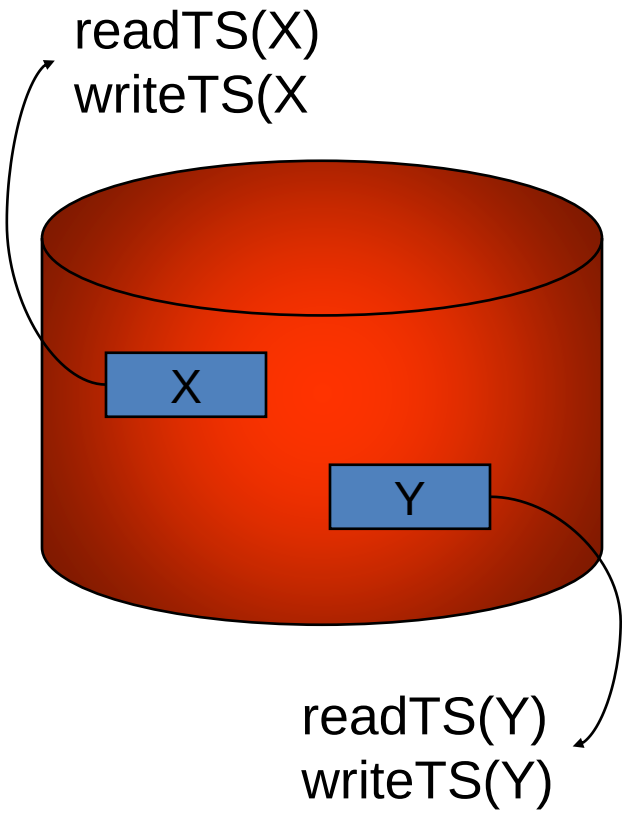
TS da transação mais recente que leu o dado

TS da transação mais recente que atualizou o dado

Ordenação por *Timestamp*

- Técnica de bloqueio
  - Ordem de execução das transações determinada pela ordem de obtenção de bloqueios sobre objetos
- Ordenação por *timestamp*
  - Ordem de execução das transações determinada pelo ordem em que as transações são iniciadas
  - *Timestamp*: Identificador único serial capaz de determinar o tempo de início de uma transação
  - Não há bloqueio → Não há *deadlocks*

Timestamp dos objetos



ReadTS: maior entre todos os *timestamps* de transações que leram o objeto

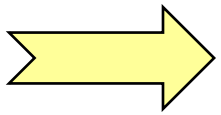
WriteTS: maior entre todos os *timestamp* de transações que modificaram o objeto

Protocolo

1. T solicita uma operação write(X)
   Se readTS(X) > TS(T) ou writeTS(X) > TS(T)
   Então abortar T
   Senão executar write(X) e writeTS(X)=TS(T)

2. T solicita uma operação read(X)
   Se writeTS(X) > TS(T)
   Então abortar T
   Senão executar read(X) e
   readTS(X)=max(readTS(X),TS(T))

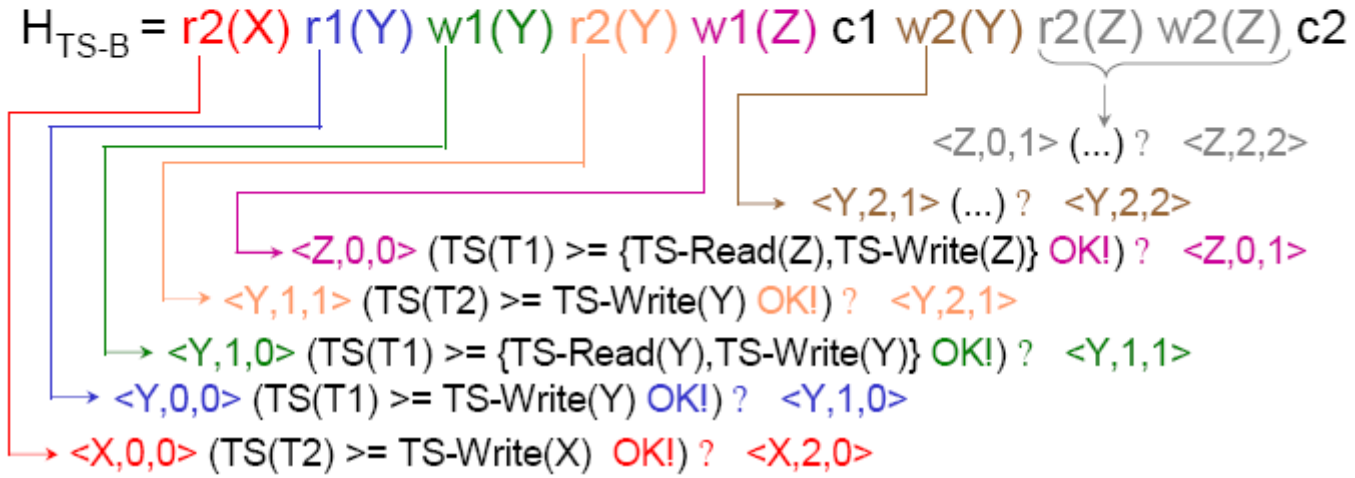
Serializabilidade: operações conflitantes sempre executadas na ordem de timestamps das transações



Possibilidade de Rollback em cascata

Técnica TS-Básico - Exemplo

- T1: r(Y) w(Y) w(Z) ? TS(T1) = 1
- T2: r(X) r(Y) w(Y) r(Z) w(Z) ? TS(T2) = 2
- Registros iniciais de TS de X, Y e Z:
  - <X,0,0>; <Y,0,0>; <Z,0,0>
- Exemplo de escalonamento serializável por TS



Técnica TS Básico

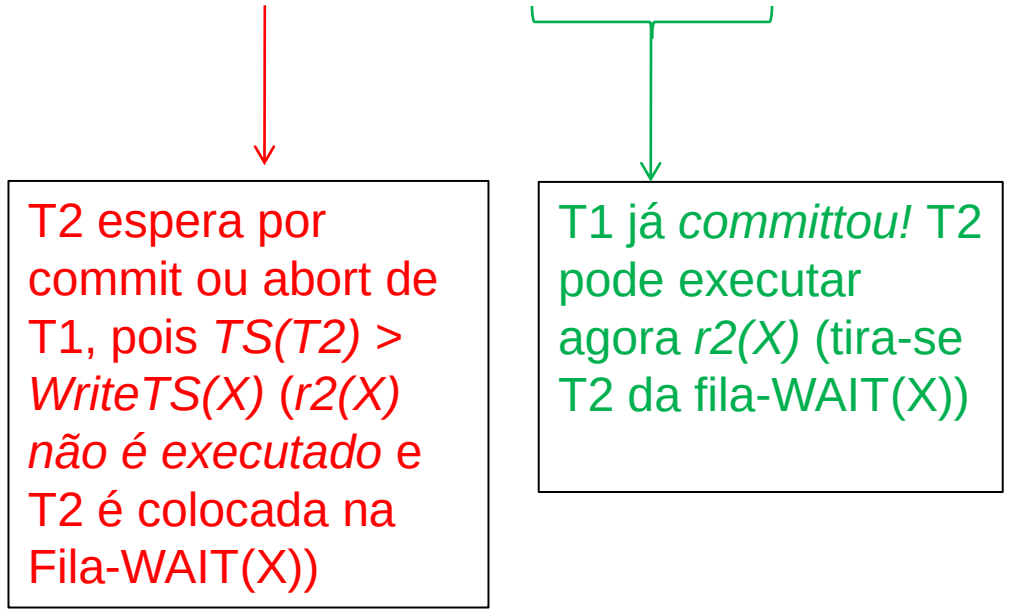
- Vantagens
  - Técnica simples para garantia de serializabilidade (não requer bloqueios)
  - Não há *deadlock* (não há espera)
- Desvantagens
  - Gera muitos abortos de transações
    - passíveis de ocorrência quando há conflito
  - Pode gerar abortos em cascata
    - não gera escalonamentos adequados ao recovery
  - Para minimizar essas desvantagens
    - técnica de *timestamp* estrito (TS-Estrito)

Técnica TS Estrito

- Garante escalonamentos serializáveis e estritos
  - Funcionamento
    - Baseado no TS-básico com a seguinte diferença:
    - “se Tx deseja read(D) ou write(D) e TS(Tx) > WriteTS(D), então Tx espera pelo commit ou abort da transação Ty cujo WriteTS(D) = TS(Ty)”
  - Exige fila-WAIT(D) – simula um bloqueio
  - Não há risco de *deadlock*
  - Nunca há ciclo pois somente transações mais novas esperam pelo commit/abort de transações mais antigas
  - *Overhead* no processamento devido à espera

Técnica TS-Estrito - Exemplo

- T1: r(X) w(X) w(Z) ? TS(T1) = 1
- T2: r(X) w(X) w(Y) ? TS(T2) = 2
- Exemplo de escalonamento TS-Estrito
- HTS-E = r1(X) w1(X) r2(X) w1(Z) c1 r2(X) w2(X) w2(Y) c2



Exercício

Considerando a técnica TS-Básico, verifique se alguma transação abaixo é desfeita e em que ponto

a) H1 = r1(a) r2(a) r3(a) c1 c2 c3
b) H2 = r1(a) w2(a) r1(a) c1 c2
c) H3 = r1(a) r1(b) r2(a) r2(b) w2(a) w2(b) c1 c2
d) H4 = r1(a) r1(b) r2(a) w2(a) w1(b) c1 c2
e) H5 = r2(a) w2(a) w1(a) r2(a) c1 c2
f) H6 = r2(a) w2(a) r1(b) r1(c) w1(c) w2(b) c1 c2

Dúvidas?