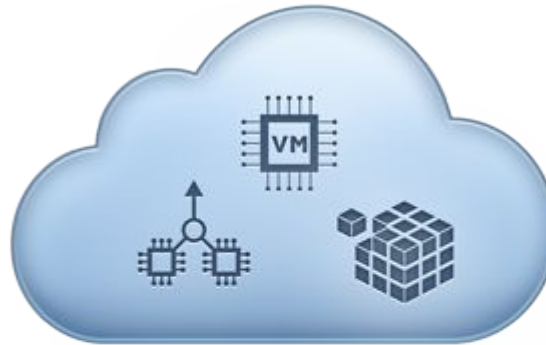


Computação em nuvem

Tecnologias de Suporte à Computação em Nuvem



Prof. Dr. Marcos A. Simplicio Jr.
Laboratório de Arquitetura e Redes de Computadores
Departamento de Engenharia de Computação e
Sistemas Digitais
Escola Politécnica da Universidade de São Paulo

Objetivos – Aula 19

- Entender os problemas gerados por **paralelismo e concorrência** entre processos, bem como soluções para os mesmos

Concorrência e paralelismo

- ❑ Sistemas distribuídos devem permitir **acesso concorrente**, mas de forma a **garantir a integridade** dos dados
- ❑ Transações: conjunto de ações que devem ser executadas de forma **atômica** (exemplo clássico: transações bancárias)
→ **Sem interferência** de outras transações executando em paralelo





Thread1: depositar 50	Thread2: depositar 200	Saldo
saldoAtual = getSaldo(user) :\$100		\$100
	saldoAtual = getSaldo(user) :\$100	
	novoSaldo = saldoAtual + 200 :\$300	
novoSaldo = saldoAtual + 50 :\$150		
	setSaldo(novoSaldo)	\$300
setSaldo(novoSaldo)		\$150
getSaldo(user) :\$150	getSaldo(user) :\$150	

Sofreu interferência ☹

Concorrência e paralelismo: locks

❑ Semáforos ("locks"):

- Acesso serializado a objetos por diferentes threads: primeiro thread **bloqueia** seguintes durante transação
- Estratégia comum para evitar inconsistências

Thread1: depositar 50	Thread2: depositar 200	Saldo
 saldoAtual = getSaldo(user) :\$100	<...aguarda...>	\$100
novoSaldo = saldoAtual + 50 :\$150	<...aguarda...>	
setSaldo(novoSaldo)	<...aguarda...>	
 getSaldo(user) <u>:\$150</u>	<...aguarda...>	\$150
 saldoAtual = getSaldo(user) :\$150		
	novoSaldo = saldoAtual + 200 :\$350	
	setSaldo(novoSaldo)	\$350
	getSaldo(user) <u>:\$350</u>	

Concorrência e paralelismo: locks



❑ Problemas com locks

- Locks **reduzem** potencial de **paralelismo**: algumas (partes de) transações poderiam ser executadas sem interferir com outras
 - Ex.: os vários threads apenas lendo
 - Abordagem de **pior caso**: mesmo transações apenas de leitura precisam usar locks, algo desnecessário (**leituras** não causam inconsistência na base de dados!)
- **Custo computacional** relativo aos locks (obtenção, aviso de liberação, ...)
 - Locks podem levar a situações de **deadlock** (um thread bloqueia outro e nenhum pode prosseguir), exigindo controle adicional (detecção, timeouts, etc)

Deadlock ilustrado



São Paulo, 2013: Av. Faria Lima com Av. Juscelino Kubitschek

Concorrência e paralelismo: conflitos

- Vamos analisar: **o que causa conflitos?**
 - **No exemplo:** operações de **escrita** baseadas em **valor lido desatualizado**
 - **Genericamente:** duas operações concorrentes cujo **resultado depende da ordem** em que elas são executadas
 - **Leitura/Leitura:** não causa conflitos
 - **Leitura/Escrita e Escrita/Escrita:** causam conflitos
- Controle otimista de concorrência (OCC)
 - Premissa: na maioria das aplicações, é **baixa a probabilidade** de que dois clientes acessem o mesmo objeto simultaneamente
 - Logo: vale mais a pena **corrigir conflitos** (abortar transações) quando eles ocorrerem do que prevenir sua ocorrência.

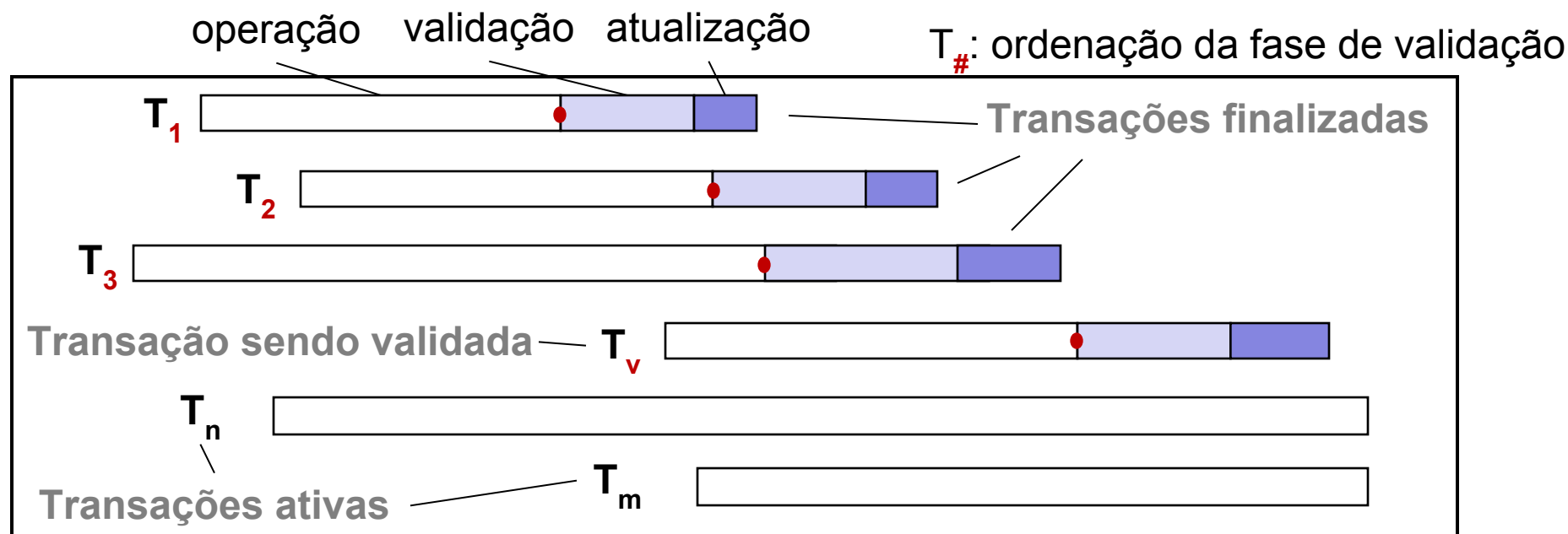


Concorrência e paralelismo: OCC



- ❑ Controle otimista de concorrência: fases
 - Operação: cada transação recebe uma **cópia local** (valor mais atual possível) dos objetos que ela deseja modificar, e prosseguem como se conflitos fossem impossíveis de ocorrer;
 - Validação: quando a transação é finalizada ("fimTransação"), verifica-se se houve algum **conflito (interferência)** entre transações), **abortando**-se um número suficiente de transações conflitantes para resolver o conflito;
 - Atualização: se a transação é válida, todas as mudanças feitas na versão local do objeto tornam-se **permanentes**.

Concorrência e paralelismo: OCC



T_v	T_i	Regra	Validação
leitura	leitura	0	Sem problemas ✓
escrita	leitura	1	$T_{i > v}$ não deve ler objetos escritos por T_v
leitura	escrita	2	T_v não deve ler objetos escritos por $T_{i < v}$
escrita	escrita	3	T_i não deve escrever objetos escritos por T_v e vice-versa → Sem problemas <u>se atualizações não se sobrepõem</u> ✓

Concorrência e paralelismo: OCC



Validação : duas abordagens possíveis

➤ Backward: análise das transações passadas que apresentem alguma sobreposição com T_v

- Regra 1: " $T_{i > v}$ não deve ler objetos escritos por T_v "

✓ – Obs.: a validação das transações futuras se preocupará com T_v

- Regra 2: " T_v não deve ler objetos escritos por $T_{i < v}$ "

✗ – Se isto ocorre, T_v leu algum valor desatualizado: T_v é abortado

✓ – Caso contrário, T_v é validado

Concorrência e paralelismo: OCC

Validação : duas abordagens possíveis

➤ Forward: análise das transações ainda ativas

- Regra 2: " T_v não deve ler objetos escritos por $T_{i < v}$ "

✓ – T_v não é afetado por escritas futuras...

- Regra 1: " $T_{i > v}$ não deve ler objetos escritos por T_v " → isto significa que T_v pode causar leituras errôneas em transações em andamento.

Opções:



- Postergar validação até que transações ativas conflitantes finalizem (afinal, elas podem ser abortadas por usuário ou falhar).

» Obs.: esta espera pode levar ao aparecimento de novas transações ativas conflitantes...



- Abortar transações conflitantes em andamento e validar T_v



- Abortar T_v



Resumo

- ❑ Entender os problemas gerados por paralelismo e concorrência entre processos, bem como soluções para os mesmos
 - Concorrência pode causar **interferências indesejadas**
 - **Locks** (semáforos): previnem interferências mas têm impacto negativo em desempenho (**reduz poder de paralelismo**)
 - **Controle Otimista de Concorrência** (OCC): assume que operações não interferem entre si, corrigindo situações indesejadas caso necessário (**melhor potencial de paralelismo**)
- ❑ Próxima aula: Big Data e MapReduce

