Controle de Concorrência

com

Locks

Locks

- Método para controlar acesso concorrente a dados compartilhados em sistemas de banco de dados através de **transações**.
- **Locks** tem sido usados em SGBDs por muito anos.
- Serviço de Controle de Concorrência do CORBA é inteiramente baseado no uso de Locks.

Locks

- Esta unidade discute a **aplicação de transações e controle de concorrência** para **objetos compartilhados** gerenciados por servidores.

Conceito de Transação

- **Transações** podem ser vistas como um grupo de operações combinadas em uma unidade lógica de trabalho.

- São usadas para controlar e manter a **consistência** e a **integridade** de cada ação em uma transação, a despeito dos erros que poderão ocorrer no sistema.

Conceito de Transação

- Uma **transação** define uma **sequência de operações que é garantida por um servidor**, para ser **atômica** na presença de **múltiplos clientes** e na classe de **falhas por *crash*** de **processos** em servidores.

Atomicidade de Transações

- **A**tomicidade: “uma transação deve ser tudo ou nada”.
- **C**onsistência: “uma transação toma o sistema de um estado consistente para um outro estado consistente”.
- **I**solamento: “cada transação deve ser realizada sem interferência de outras transações”.
- **D**urabilidade: “após uma transação ter sido completada bem sucedida, todos os seus efeitos são salvos em memória permanente.

Classe de Falha: Crash

- Afeta: Processo
- Descrição: O processo pára e permanece parado. Outros processos podem **não** ser capazes de detectar este estado.

Equivalência Serial

- Todos os protocolos de controle de concorrência são baseados sobre o critério de **equivalência serial** e são derivados de regras de **conflitos entre operações**.

Locks

- ***Locks*** são usados para ordenar transações que acessam os mesmos objetos de acordo com a ordem de chegada de suas operações nos objetos.

Meta de Transações

- Garantir que todos os objetos gerenciados por um servidor permaneçam em um estado consistente quando eles são acessados por múltiplas transações e na presença de falha de *crash* dos servidores.

- Discutimos questões para **transações sobre um único servidor**.

Indivisibilidade

- Uma transação de cliente é também considerada como **indivisível do ponto de vista da transação de outro cliente**, no sentido que **as operações de uma transação não podem observar os efeitos parciais das operações de uma outra transação**.

Um exemplo

- Usamos como exemplo, **uma aplicação bancária**.

- Cada **conta** é representada por **um objeto remoto**, cuja interface ***Account*** provê operações para fazer **depósito**, **saques**, **estabelecer** (calcular) **saldos** e pedir informações sobre esses.

Operations of the *Account* interface

- *deposit(amount)*
  - deposit amount in the account
- *withdraw(amount)*
  - withdraw amount from the account
- *getBalance() -> amount*
  - return the balance of the account
- *setBalance(amount)*
  - set the balance of the account to amount

Agências do Banco

- Cada agência é representada por um objeto remoto cuja interface *Agency* provê operações para **criar uma nova conta**, **procurar uma conta por nome** e pedir informações do total de fundos naquela agência.

Operations of the *Agency* interface

- *create(name) -> account*
  - create a new account with a given name.
- *lookUp(name) -> account*
  - return a reference to the account with the given name.
- *agencyTotal() -> amount*
  - return the total of all the balances at the agency.

Sem sincronização

- Servidores não cuidadosamente projetados, suas operações em nome de diferentes clientes podem algumas vezes interferir com outras operações.

- Tais interferências podem resultar em valores incorretos nos objetos.

Com sincronização

- Operações de clientes podem ser sincronizadas sem o recurso de transação.

- Métodos de objetos devem ser projetados para uso em um contexto ***multi-threaded***.

- Métodos *synchronized* do Java.

Com sincronização

- Operações que são **livres de interferência** de operações concorrentes sendo realizadas em outras threads são chamadas **operações atômicas**.

- O uso de métodos synchronized em Java é um modo de alcançar operações atômicas.

Transações

- Existem situações, que clientes requerem uma sequência de solicitações separadas para um servidor, e a sequência deve ser atômica no sentido que:

Transações

1. As operações dentro da sequência são livres de interferência de operações sendo realizadas em nome de outros clientes concorrentes;

Transações

2. Ou todas as operações na sequência devem ser completadas bem sucedidas ou elas devem ter nenhum efeito sobre tudo na presença de falha do servidor por *crash*.

Uma transação T de um cliente

- *Transaction T:*

  a.withdraw(100);

  b.deposit(100);

  c.withdraw(200);

  b.deposit(200);

- Supõem-se contas com nomes *A, B, C e* variáveis *a, b, c* do tipo *Account.*

Transações

- Originam-se de um SGBD.

- Uma transação é a execução de uma sequência de solicitações (*requests*) de um cliente sobre operações (*withdraw*, *deposit*).

Transação

- Do ponto de vista do cliente, **uma transação é uma sequência de operações que formam um única etapa**, transformando os dados de um servidor de **um estado consistente** para **um outro estado consistente**.

- O cliente é provido com operações para marcar o **início e o fim de uma transação**.

Middleware CORBA

- Object Transaction Service [OMG 2003].

- Interfaces IDL permitem transações de clientes incluem múltiplos objetos em múltiplos servidores.

Middleware CORBA

- O ORB cliente mantém **um contexto para cada transação**, a qual ele propaga, **com cada operação** na transação.

- **Objetos transacionais** são invocados **dentro do escopo de uma transação** e geralmente têm alguma **armazenagem persistente** associados com eles.

Objetos Recuperáveis

- Objetos que podem ser recuperados **após uma falha por *crash*** de servidores.

- Em geral, objetos gerenciados por um servidor podem ser armazenados em memória volátil (RAM) ou memória persistente (hard disk).

Objetos Recuperáveis

- Em todo contexto, uma transação aplica-se a objetos recuperáveis e é voltada para ser atômica.

- Frequentemente chamada de transação atômica:

Primeiro aspecto para a atomicidade

“Tudo ou nada”

Uma transação ou é completada bem sucedidamente e o efeito de todas as suas operações é registrado nos objetos, ou se ela falha ou é deliberadamente abortada, ela não tem nenhum efeito no todo.

Aspectos adicionais deste aspecto

Atomicidade de falha: os efeitos são atômicos mesmo quando os servidores falham em crash.

Durabilidade: após uma transação ter sido completada bem sucedidamente, todos os seus efeitos são salvos em memória permanente. Dados salvos em mídia permanente, sobreviverão mesmo se o processo servidor falhar por crash.

Segundo aspecto para atomicidade

- Isolamento:

  Cada transação deve ser realizada sem interferência de outra transação.

Requisitos

- Para suportar os requisitos de atomicidade de falha e durabilidade, os objetos devem ser recuperáveis.

- Para garantir o aspecto de Isolamento, o servidor que suporta transações deve sincronizar as operações suficientemente.

Garantir Isolamento

- Um modo de fazer isto é realizar transações serialmente.

- Mas, no caso de servidores que compartilham recursos para muitos usuários interativos ?

Garantir Isolamento

- Em um banco real é desejável que os caixas realizem transações bancárias on-line ao mesmo tempo.

- Para resolver, o objetivo para qualquer servidor que suporte transações é maximizar concorrência.

Concorrência de Transações

- Transações são permitidas executarem concorrentemente, se elas têm o mesmo efeito como na execução serial.

- São serialmente equivalentes.

Coordenador de Transações

- Cada transação é criada e gerenciada por um coordenador, o qual implementa uma interface ***Coordinator***.

Operações na Interface *Coordinator*

- *openTransaction() -> trans;*
  - starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

- *closeTransaction(trans) -> (commit, abort);*
  - ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

- *abortTransaction(trans);*
  - aborts the transaction.

Transaction life histories

Successful	*Aborted by client*	*Aborted by server*	
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
⋮	⋮	server aborts transaction →	⋮
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

Controle Concorrência

- Dois problemas bem conhecidos de transações concorrentes no contexto do exemplo do banco:
  - "lost update"
  - "inconsistent retrivals"

- Como estes problemas podem ser evitados usando-se **equivalência serial de execuções** de transações ?

Operações

- Assuminos que cada das operações *deposit*, *withdraw*, *getBalance*, *setBalance*, é uma *synchronized operação*, isto é, seus efeitos sobre a variável de instância que registra o *balance* (saldo) de uma conta é atômico.

O problema “lost update”

- Sejam as contas A, B e C.
- Sejam duas transações T e U sobre as contas A, B e C.
- Os valores iniciais de *balance* são:
  - A igual a $100,
  - B igual a $200,
  - C igual a $300.

O problema “lost update”

- A transação T transfere um valor da conta A para a conta B.
- A transação U transfere um valor da conta C para a conta B.
- Em ambos os casos, o valor transferido é calculado para aumentar o saldo (balance) de B em 10%.

Observação das Figuras

- Daqui para frente, são mostradas as operações que afetam a variável balance (saldo) de uma conta, nas sucessivas linhas das seguintes figuras, e o leitor deve assumir que uma operação, numa linha em particular, é executada num tempo posterior do que a linha acima.

The "lost update" problem

Transaction : *T*		Transaction : *U*	
balance= b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance= b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance=b.getBalance();	$200		
		balance= b.getBalance()	$200
		*b.setBalance(balance*1.1)*	$220
*b.setBalance(balance*1.1)*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Resultado Correto!

- O efeito sobre a conta B de executar as transações T e U, deve ser para aumentar o *balance* (saldo) de B em 10%, duas vezes. Assim, o valor final deveria ser $242.

Resultado !

- Os efeitos de permitir as transações T e U rodarem concorrentemente como na figura "lost update", ambas as transações obtém o *balance* de B como $200 e então *deposit* $20.

- O resultado é incorreto, aumentando o *balance* de B em $20 ao invés de $42.

Por que ?? Erro !!!

- O “update” de U é perdido porque T sobrescreve *balance* de B sem ver o “update” de U.

- Ambas as transações tem de ler o valor inicial de *balance* de B, antes de qualquer deles escrever o novo valor de *balance* de B.

The “lost update” problem

- O problema de “lost update” ocorre quando duas transações T e U lêem o valor velho de uma variável (balance) e então usa ele para calcular o novo valor dessa variável (balance).

The “lost update” problem

- Isto não pode acontecer, se uma transação é realizada antes da outra, porque a última transação lerá o valor escrito pela última transação.

Resolvendo “lost update”

- Pode-se resolver o problema “lost update” por meio de uma **equivalência serial de intercalações de transações T e U**.

A serially equivalent interleaving of *T* and *U*

T Transaction	U Transaction
balance = b.getBalance()	*balance = b.getBalance()*
*b.setBalance(balance*1.1)*	*b.setBalance(balance*1.1)*
a.withdraw(balance/10)	*c.withdraw(balance/10)*
balance = b.getBalance() $200	
*b.setBalance(balance*1.1)* $220	
	balance = b.getBalance() $220
	*b.setBalance(balance*1.1)* $242
a.withdraw(balance/10) $80	
	c.withdraw(balance/10) $278

A serially equivalent interleaving of *T* and *U*

- A figura anterior mostra uma intercalação na qual as operações que afetam uma conta compartilhada, B, são realmente seriais.

- Ou seja, a transação T faz todas as suas operações sobre B, antes da transação U fazer.

A serially equivalent interleaving of *T* and *U*

- Uma outra intercalação de T e U que tem esta propriedade é uma na qual a transação U completa suas operações sobre a conta B, antes da transação T iniciar.

The Inconsistent Retrievals Problem

Transaction : *V*		**Transaction : *W***	
a.withdraw(100) *b.deposit(100)*		*aAgency.agencyTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

The Inconsistent Retrievals Problem

- Um outro exemplo de problema relacionado a uma conta bancária.

- A transação V transfere a soma das contas A e B e a transação W invoca o método *agencyTotal* para obter a soma dos saldos de todas as contas numa agência do banco.

The Inconsistent Retrievals Problem

- Os saldos (*balance*) das duas contas A e B são ambos inicialmente $200,00.
- O resultado de *agencyTotal* inclui a soma de A e B como $300,00, o que é errado.
- Isto ilustra o problema de Inconsistent Retrivals.

The Inconsistent Retrievals Problem

- *Retrivals* (recuperações) de W são inconsistentes porque a transação V realizou somente a parte de saque (withdrawal) de uma transferência no tempo em que a soma é calculada.

A serially equivalent interleaving of *V* and *W*

Transaction : *V*		**Transaction : *W***	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

A serially equivalent interleaving of *V* and *W*

- Considere agora, o efeito de da equivalência serial em relação ao problema “inconsistent retrivals”, no qual a transação V está transferindo a soma da conta A para B, e a transação W está obtendo a soma de todos os saldos.

A serially equivalent interleaving of *V* and *W*

- O “inconsistent retrivals problem” pode ocorrer quando uma transação de recuperação executa concorrentemente com outra transação de “update”.

A serially equivalent interleaving of *V* and *W*

- O problema *"inconsistent retrivals"* não pode ocorrer se uma transação de recuperação é executada antes ou após a transação de "*update"* ocorrer.

A serially equivalent interleaving of *V* and *W*

- Uma intercalação de equivalência serial de uma transação W de recuperação (“retrieval”) e uma transação V de atualização (“update”), impede de ocorrer recuperações inconsistentes.

A serially equivalent interleaving of *V* and *W*

Transaction : *V*		**Transaction : *W***	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

Equivalência Serial

- Se cada das **diversas transações** é conhecida ter o efeito correto quando ela é feita sobre o que é próprio dela, então podemos inferir que se estas transações são feitas uma em um tempo, em alguma ordem, o efeito combinado será também correto.

Equivalência Serial

- Uma intercalação das operações de transações na qual o efeito combinado é o mesmo como se as transações tivessem sido executadas uma em um tempo, em alguma ordem, é considerada uma intercalação equivalente serialmente.

Equivalência Serial

- Dizemos que **duas transações diferentes têm o mesmo**, quando as operações de leitura retornam os mesmos valores, e as variáveis de instância dos objetos têm, no final, o mesmo valor.

Equivalência Serial

- O uso de equivalência serial como um critério para execução concorrente correta de transações, impede a ocorrência de atualizações perdidas (“lost updates”) e recuperações inconsistentes (“inconsistent retrievals).

Operações Conflitantes

- Pares de operações são confitantes, se seus efeitos combinados depende da ordem na qual a operações no par são executados.

- Considerando um par *read* e *write*, a operação *read* acessa o valor de um objeto e *write* muda seu valor.

Operações Conflitantes

- O ***efeito*** de um operação refere-se ao valor de um objeto estabelecido por uma operação *write* e o resultado retornado por uma operação *read*.

- As regras de conflito para as operações read e write são dadas no slide que segue:

Read and *write* operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed.
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution.
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution.

Operações Conflitantes

- Para quaisquer par de transações, é possivel determinar a ordem de pares de operações conflitantes, sobre objetos acessados por ambas as transações.

Operações Conflitantes

- Equivalência serial pode ser definida em termos de conflitos de operações como segue:

  "Para duas transações serem **equivalentes serialmente**, é necessário e suficiente que **todos os pares de operações conflitantes das duas transações sejam executados na mesma ordem**, em todos os objetos que as transações acessam".

(10) A non-serially equivalent interleaving of operations of transactions *T* and *U*

Transaction : *T*	Transaction : *U*
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Intercalação Não-Serialmente Equivalente de operações de Transações T e U

- Considere a figura em (10), com as transações T e U definidas.

- Então considere a intercalação de suas execuções como em (10).

- Note que cada acesso de transação aos objetos *i* e *j* é serializado com respeito a um outro.

Intercalação Não-Serialmente Equivalente de operações de Transações T e U

- Porque T faz todos os seus acessos a i antes de U fazer e U faz todos os seus acessos a j antes de T fazer.

- Porém, a ordem não é serialmente equivalente, porque os pares de operações conflitantes não são feitos na mesma ordem em ambos os objetos.

Intercalação Não-Serialmente Equivalente de operações de Transações T e U

- **Ordens serialmente equivalentes** requerem **uma** das seguintes condições:
  - T acessa *i* antes de U e T acessa *j* antes de U.
  - U acessa *i* antes de T e U acessa *j* antes de T.

Locks

- Transações devem ser escalonadas de modo que seus efeitos sobre dados compartilhados sejam equivalente serialmente.

Como se pode serializar transações

- Um servidor pode alcançar equivalência serial de transações por serializar o acesso aos objetos.

- Ver código (7) que mostra como equivalência serial pode ser obtida com algum grau de concorrência ...

(7)

A serially equivalent interleaving of *T* and *U*

T Transaction	U Transaction
balance = b.getBalance()	*balance = b.getBalance()*
*b.setBalance(balance*1.1)*	*b.setBalance(balance*1.1)*
a.withdraw(balance/10)	*c.withdraw(balance/10)*

balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Locks

- Sejam as transações T e U.
- Ambas acessam uma conta B, mas T completa seu acesso antes de U iniciar acesso à conta B.
- Um exemplo simples de um mecanismo de serialização é o uso de *locks* exclusivos.

Locks

- Neste esquema, um servidor tenta bloquear qualquer objeto que é para ser usado por qualquer operação de uma transação de cliente.

Locks

- Se um cliente requer acesso a um objeto que já está bloqueado, devido a uma outra transação de cliente, a requisição é suspensa e o cliente deve esperar até que o objeto seja desbloqueado.

Transactions *T* and *U* with exclusive locks

Transaction: *T*		**Transaction: *U***	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
		• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
closeTransaction	unlock *A*, *B*	*closeTransaction*	unlock *B* *C*

Locks

- Neste exemplo é assumido que quando as transações T e U iniciam, os saldos (balances) das contas A, B e C ainda não estão bloqueados.

- Quando a transação T é para usar a conta B, esta é bloqueada para T.

Locks

- Quando a transação U é para usar a conta B, B ainda está bloqueada para T, e a transação U espera (wait).

- Quando a transação T é consolidada (**committed**), B é desbloqueada, onde, então, a transação U é retomada.

Locks

- O uso do *lock* sobre B serializa o acesso a B.

- Por exemplo, se T tivesse liberado o lock sobre B, entre: *bal=b.getBalance()* e b.*setBalance(bal*1.1)*

  a operação *bal=getBalance()* da transação U, sobre B poderia ser intercalada entre elas.

Locks

- Equivalência serial com Locks, requer que todos os acessos de uma transação para um particular objeto, seja serializado com respeito aos acessos por outras transações.

- Todos os pares de operações conflitantes de duas transações, devem se executadas na mesma ordem.

Locks

- Para garantir isto, não é permitido quaisquer novos *locks*, após a transação ter liberado um *lock*.

- A primeira fase de uma transação é uma durante a qual novos *locks* podem ser adquiridos (acquired).
- Na segunda fase, os *locks* são liberados.

Exemplo de Transação

1. Você esta fazendo uma transferência da poupança para a conta corrente.
2. Ao realizar o saque da poupança a transação é iniciada.
3. A transação é vista como o grupo de operações : saque poupança => credito conta corrente.
4. O saque da poupança é efetuado com sucesso , mas na hora de creditar a conta corrente o sistema caiu...
5. A transação não foi completada pois o crédito do dinheiro na conta não foi efetuado.
6. O sistema desfaz a operação de saque da poupança (credita novamente o valor) para não haver inconsistência na transação.

O Processamento de Transações

- BeginTrans, CommitTrans, RollBack, CloseTrans, AbortTrans,

- Um objeto Transaction é usado para **consolidar** (**commit**) ou **desfazer** (**rollback**) as modificações realizadas na fonte de dados com base no sucesso ou não dos componentes da transação.

(16) Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

---

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Implementando Locks

- *Locks* são implementados por um objeto separado no lado do servidor, que chamamos de ***Lock Manager*** *(Gerenciador de Locks).*

- É implementado por uma classe chamada *LockManager.*

- *LockManager* retém um conjunto de *locks*, por exemplo em uma tabela *hash*.

Implementando Locks

- Cada *lock* é uma instância da classe *Lock* e é associada com um particular objeto.

- Ver a implementação de uma classe *Lock*.

Lock class

```
public class Lock {
        private Object object;          // the object being protected by the lock
        private Vector holders;         // the TIDs of current holders
        private LockType lockType;      // the current type
        public synchronized void acquire(TransID trans,  LockType  aLockType ){
                while(/*another transaction holds the lock in conflicing mode*/) {
                        try {
                                wait();
                        }catch ( InterruptedException e){/*...*/ }
                }
                if(holders.isEmpty()) {  // no TIDs  hold lock
                        holders.addElement(trans);
                        lockType  = aLockType;
                } else if ( /*another transaction holds the lock, share it*/ ) ) {
                    if(/* this transaction not a holder*/) holders.addElement(trans);
                } else if (/* this transaction is a holder but needs a more exclusive
                        lock*/)
                        lockType.promote();
                }
        }
```

Continues on next slide

Class Lock (continued)

```
public synchronized void release(TransID trans ){
                holders.removeElement(trans);     // remove this holder
                // set locktype to none
                notifyAll();
        }
}
```

Implementação de Locks

- Cada instância de Lock mantém a seguinte informação em suas variáveis de instância:
  - O identificador do objeto bloqueado.
  - Os identificadores de transações que correntemente retém o *lock* (*Locks* compartilhados podem ter diversos retentores – *holders*.
  - Um tipo de *lock*.

Implementação de Locks

- Os métodos da classe *Lock* são sincronizados, de modo que as threads (transações) tentando adquirir (estabelecer) ou liberar um lock, não interferirá com uma outra thread.

- Tentativas para adquirir um *lock* usam o método *wait* sempre que uma transação tem que esperar para uma outra thread liberar o *lock* (bloqueio).

LockManager

- A classe LockManager é mostrada no slide seguinte.

- Todos as requisições para estabelecer Locks e para liberar locks em nome de transações, são enviadas para uma instância de *LockManager*.

Lock Manager

```
public class LockManager {
   private Hashtable theLocks;

   public  void setLock(Object object, TransID trans,  LockType lockType) {
      Lock foundLock;
      synchronized(this){
               // find the lock associated with object
               // if there isn't one, create it and add to the hashtable
               }
      foundLock.acquire(trans, lockType);
   }
  // synchronize this one because we want to remove all entries
  public synchronized void unLock(TransID trans) {
      Enumeration e = theLocks.elements();
      while(e.hasMoreElements()){
         Lock aLock = (Lock)(e.nextElement());
         if (/* trans is a holder of this lock*/ ) aLock.release(trans);
      }
   }
}
```