

TRABALHO PRÁTICO 4

Sérgio Mergen

Código Fonte

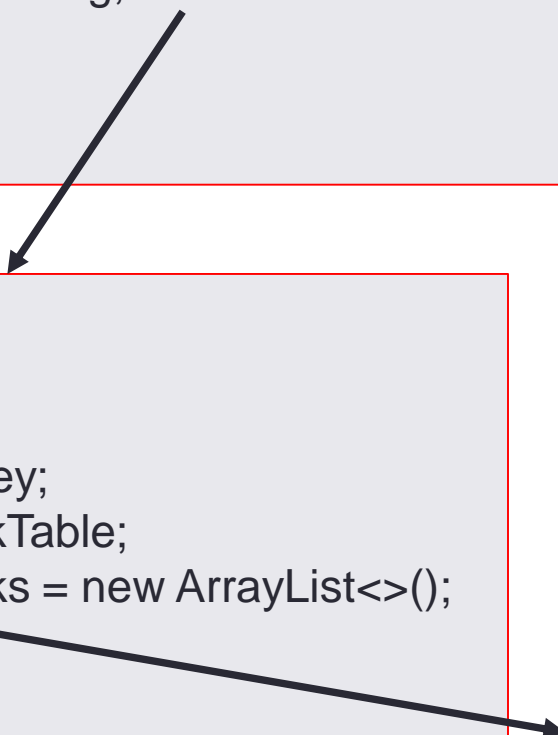
- O SGBD criado para a disciplina tem suporte a algumas operações de acesso aos dados
 - Transaction: uma transação
 - Instruction: uma instrução de uma transação
 - Item: um item de dados (um registro de uma tabela)
 - Lock: uma transação na fila de espera para acessar um item
 - LockTable: a tabela de bloqueios de todos os itens
 - ConcurrencyManager: o gerenciador de acesso concorrente
 - SimulatedIterations: simula um ambiente de concorrência composto por múltiplas transações

Classe LockTable

- O gerenciador de acesso concorrente usa uma tabela de locks (LockTable) para decidir se uma transação pode obter acesso a um item de dados

Classe LockTable

```
public class LockTable {  
    private Hashtable<String, Item> itens = new Hashtable<>();  
  
    ...  
}
```



```
public class Item {  
  
    public Table table;  
    public long primaryKey;  
    public LockTable lockTable;  
    ArrayList<Lock> locks = new ArrayList<>();  
  
    ...  
}
```

```
public class Lock {  
    Transaction transaction;  
    int mode;  
  
    ...  
}
```

ConcurrencyManager


- A função **processNextInstruction** é responsável por executar a próxima instrução de uma transação recebida por parâmetro
- O retorno é o registro (lido no caso de READ) ou gerado (no caso do WRITE)
- Caso a instrução não possa ser executada, o retorno é nulo

```
public Record processNextInstruction(Transaction t) throws Exception {  
    if (!t.waitingLockRelease()) {  
        Transaction toAbort = lockTable.queueTransaction(t.getCurrentInstruction());  
        if (toAbort!=null)  
        {  
            abort(toAbort);  
            return null;  
        }  
    }  
    if (t.canLockCurrentInstruction())  
        return t.processCurrentInstruction();  
    return null;}  
}
```

ConcurrencyManager

- A função **processNextInstruction** é responsável por executar a próxima instrução de uma transação recebida por parâmetro
- O retorno é o registro (lido no caso de READ) ou gerado (no caso do WRITE)
- Caso a instrução não possa ser executada, o retorno é nulo

```
public Record processNextInstruction(Transaction t) throws Exception {  
    if (!t.waitingLockRelease()) {  
        Transaction toAbort = lockTable.queueTransaction(t.getCurrentInstruction());  
        if (toAbort != null)  
        {  
            abort(toAbort);  
            return null;  
        }  
    }  
    if (t.canLockCurrentInstruction())  
        return t.processCurrentInstruction();  
    return null;  
}
```




Caso a transação não esteja bloqueada esperando a liberação de algum item

ConcurrencyManager

- A função **processNextInstruction** é responsável por executar a próxima instrução de uma transação recebida por parâmetro
- O retorno é o registro (lido no caso de READ) ou gerado (no caso do WRITE)
- Caso a instrução não possa ser executada, o retorno é nulo

```
public Record processNextInstruction(Transaction t) throws Exception {  
    if (!t.waitingLockRelease()) {  
        Transaction toAbort = queueTransaction(t.getCurrentInstruction());  
        if (toAbort!=null)  
        {  
            abort(toAbort);  
            return null;  
        }  
    }  
    if (t.canLockCurrentInstruction())  
        return t.processCurrentInstruction();  
    return null;}  

```



Solicita inclusão na tabela de bloqueios

ConcurrencyManager

- A função **processNextInstruction** é responsável por executar a próxima instrução de uma transação recebida por parâmetro
- O retorno é o registro (lido no caso de READ) ou gerado (no caso do WRITE)
- Caso a instrução não possa ser executada, o retorno é nulo

```
public Record processNextInstruction(Transaction t) throws Exception {  
    if (!t.waitingLockRelease()) {  
        Transaction toAbort = queueTransaction(t.getCurrentInstruction());  
        if (toAbort!=null)  
        {  
            abort(toAbort);  
            return null;  
        }  
    }  
    if (t.canLockCurrentInstruction())  
        return t.processCurrentInstruction();  
    return null;}  

```


Aborta transação escolhida, caso seja necessário

Retorna null indicando que não foi possível executar

ConcurrencyManager

- A função **processNextInstruction** é responsável por executar a próxima instrução de uma transação recebida por parâmetro
- O retorno é o registro (lido no caso de READ) ou gerado (no caso do WRITE)
- Caso a instrução não possa ser executada, o retorno é nulo

```
public Record processNextInstruction(Transaction t) throws Exception {  
    if (!t.waitingLockRelease()) {  
        Transaction toAbort = queueTransaction(t.getCurrentInstruction());  
        if (toAbort != null)  
        {  
            abort(toAbort);  
            return null;  
        }  
    }  
    if (t.canLockCurrentInstruction())  
        return t.processCurrentInstruction();  
    return null;  
}
```



Transação já está na fila
Verifica se pode bloquear o item

ConcurrencyManager

- A função **processNextInstruction** é responsável por executar a próxima instrução de uma transação recebida por parâmetro
- O retorno é o registro (lido no caso de READ) ou gerado (no caso do WRITE)
- Caso a instrução não possa ser executada, o retorno é nulo

```
public Record processNextInstruction(Transaction t) throws Exception {  
    if (!t.waitingLockRelease()) {  
        Transaction toAbort = queueTransaction(t.getCurrentInstruction());  
        if (toAbort!=null)  
        {  
            abort(toAbort);  
            return null;  
        }  
    }  
    if (t.canLockCurrentInstruction())  
        return t.processCurrentInstruction();  
    return null;}  

```

Retorna o registro lido ou gravado




ConcurrencyManager

- A função **processNextInstruction** é responsável por executar a próxima instrução de uma transação recebida por parâmetro
- O retorno é o registro (lido no caso de READ) ou gerado (no caso do WRITE)
- Caso a instrução não possa ser executada, o retorno é nulo

```
public Record processNextInstruction(Transaction t) throws Exception {  
    if (!t.waitingLockRelease()) {  
        Transaction toAbort = queueTransaction(t.getCurrentInstruction());  
        if (toAbort!=null)  
        {  
            abort(toAbort);  
            return null;  
        }  
    }  
    if (t.canLockCurrentInstruction())  
        return t.processCurrentInstruction();  
    return null;  
}
```

Retorna null caso não tenha sido possível obter o bloqueio



ConcurrencyManager

- A função **queueTransaction** adiciona a transação que possui a instrução **instruction** na fila de espera para acessar o item de dados referenciado pela instrução
- O retorno é a transação que precisa ser abortada caso haja algum problema
 - Retorna null se nenhuma transação precisa ser abortada

```
public Transaction queueTransaction(Instruction instruction) {  
    Item item = lockTable.getItem(instruction);  
  
    if (!alreadyInQueue(item, instruction)) {  
        return addToQueue(item, instruction);  
    }  
    else return null;  
}
```

Concur

```
public Transaction addToQueue(Item item, Instruction instruction) {  
    Transaction t = instruction.getTransaction();  
    Lock l = new Lock(t, instruction.getMode());  
    item.locks.add(l);  
    instruction.setItem(item);  
    return null;  
}
```

- A função **queueTransaction** adiciona a transação que possui a instrução **instruction** na fila de espera para acessar o item de dados referenciado pela instrução
- O retorno é a transação que precisa ser abortada caso haja algum problema
 - Retorna null se nenhuma transação precisa ser abortada

```
public Transaction queueTransaction(Instruction instruction) {  
    Item item = lockTable.getItem(instruction);  
  
    if (!alreadyInQueue(item, instruction)) {  
        return addToQueue(item, instruction);  
    }  
    else return null;  
}
```

ConcurrencyManager

- A função abort
 - cancela a execução da transação
 - E remove a transação da tabela de bloqueios (locktable)

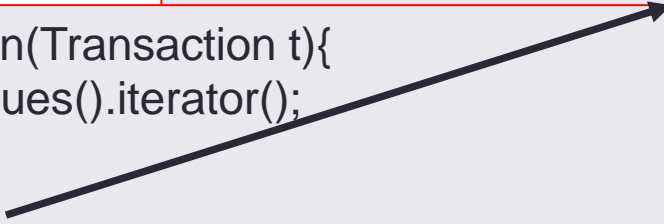
```
public void abort(Transaction t) throws Exception{  
    t.abort();  
    lockTable.removeTransaction(t);  
    activeTransactions.remove(t);  
}
```

Classe LockTable

- A função **removeTransaction** remove a transação da tabela de bloqueios
- A transação é removida de todos os itens

```
public void removeTransaction(Transaction t) {  
    for (int i = locks.size() - 1; i >= 0; i--) {  
        Lock lock = locks.get(i);  
        if (lock.transaction.equals(t)) {  
            locks.remove(i);  
        }  
    }  
}
```

```
public void removeTransaction(Transaction t){  
    Iterator<Item> it = itens.values().iterator();  
    while (it.hasNext()){  
        Item item = it.next();  
        item.removeTransaction(t);  
    }  
}
```



SimulatedIterations

- Simula o processamento intercalado de transações, nos mesmos moldes de iteração vistos em aula

```
Table tab= Utils.createTable("c:\\teste\\ibd","t1.ibd",1000, true, 1);
```

```
Transaction t1 = new Transaction();  
t1.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('A'), null));  
t1.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('B'), "b"));
```

```
Transaction t2 = new Transaction();  
t2.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('B'), null));  
t2.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('A'), "b"));
```

```
SimulatedIterations simulation = new SimulatedIterations();  
simulation.addTransaction(t1);  
simulation.addTransaction(t2);  
simulation.run(100);
```


SimulatedIterations

- Simula o processamento intercalado de transações, nos mesmos moldes de iteração vistos em aula

```
Table tab= Utils.createTable("c:\\teste\\ibd","t1.ibd",1000, true, 1);
```

```
Transaction t1 = new Transaction();  
t1.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('A'), null));  
t1.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('B'), "b"));
```

```
Transaction t2 = new Transaction();  
t2.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('B'), null));  
t2.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('A'), "b"));
```

```
SimulatedIterations simulation = new SimulatedIterations();  
simulation.addTransaction(t1);  
simulation.addTransaction(t2);  
simulation.run(100);
```

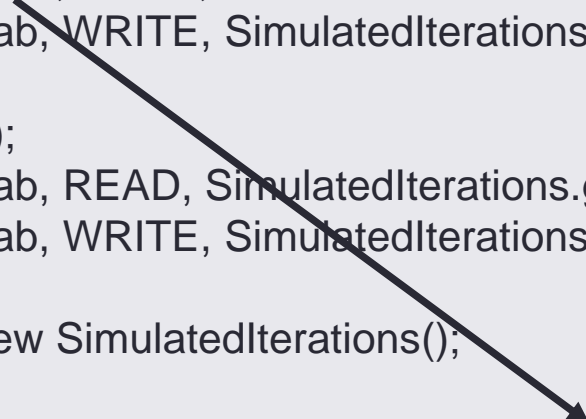


Tabela que possui o registro a ser usado

SimulatedIterations

- Simula o processamento intercalado de transações, nos mesmos moldes de iteração vistos em aula

```
Table tab= Utils.createTable("c:\\teste\\ibd","t1.ibd",1000, true, 1);
```

```
Transaction t1 = new Transaction();
```

```
t1.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('A'), null));
```

```
t1.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('B'), "b"));
```

```
Transaction t2 = new Transaction();
```

```
t2.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('B'), null));
```

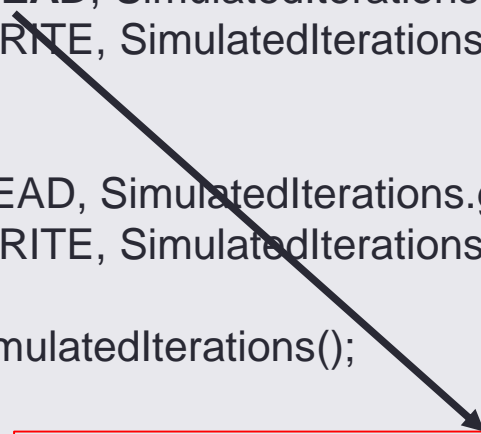
```
t2.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('A'), "b"));
```

```
SimulatedIterations simulation = new SimulatedIterations();
```

```
simulation.addTransaction(t1);
```

```
simulation.addTransaction(t2);
```

```
simulation.run(100);
```



Modo da instrução (READ ou WRITE)

SimulatedIterations

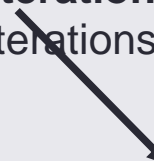
- Simula o processamento intercalado de transações, nos mesmos moldes de iteração vistos em aula

```
Table tab= Utils.createTable("c:\\teste\\ibd","t1.ibd",1000, true, 1);
```

```
Transaction t1 = new Transaction();  
t1.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('A'), null));  
t1.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('B'), "b"));
```

```
Transaction t2 = new Transaction();  
t2.addInstruction(new Instruction(tab, READ, 'A', null));  
t2.addInstruction(new Instruction(tab, WRITE, 'B', "b"));
```

```
SimulatedIterations simulation = new SimulatedIterations(tab);  
simulation.addTransaction(t1);  
simulation.addTransaction(t2);  
simulation.run(100);
```



Item (registro) a ser usado

A função `getValue` converte caractere em chave primária numérica (para ser compatível com os exemplos vistos em aula)

SimulatedIterations

- Simula o processamento intercalado de transações, nos mesmos moldes de iteração vistos em aula

```
Table tab= Utils.createTable("c:\\teste\\ibd","t1.ibd",1000, true, 1);
```

```
Transaction t1 = new Transaction();
```

```
t1.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('A'), null));
```

```
t1.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('B'), "b"));
```

```
Transaction t2 = new Transaction();
```

```
t2.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('B'), null));
```

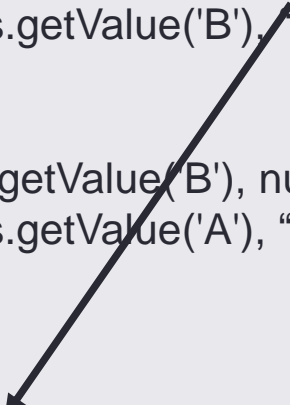
```
t2.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('A'), "b"));
```

```
SimulatedIterations simulation = new SimulatedIterations();
```

```
simulation.addTransaction(t1);
```

```
simulation.addTransaction(t2);
```

```
simulation.run(100);
```



Valor que será gravado no registro
Null caso a operação seja de leitura

SimulatedIterations


- Simula o processamento intercalado de transações, nos mesmos moldes de iteração vistos em aula

```
Table tab= Utils.createTable("c:\\teste\\ibd","t1.ibd",1000, true, 1);
```

```
Transaction t1 = new Transaction();  
t1.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('A'), null));  
t1.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('B'), "b"));
```

```
Transaction t2 = new Transaction();  
t2.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('B'), null));  
t2.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('A'), "b"));
```

```
SimulatedIterations simulation = new SimulatedIterations();  
simulation.addTransaction(t1);  
simulation.addTransaction(t2);  
simulation.run(100);
```



Transações inseridas antes são mais velhas

SimulatedIterations

- Simula o processamento intercalado de transações, nos mesmos moldes de iteração vistos em aula

```
Table tab= Utils.createTable("c:\\teste\\ibd","t1.ibd",1000, true, 1);
```

```
Transaction t1 = new Transaction();  
t1.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('A'), null));  
t1.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('B'), "b"));
```

```
Transaction t2 = new Transaction();  
t2.addInstruction(new Instruction(tab, READ, SimulatedIterations.getValue('B'), null));  
t2.addInstruction(new Instruction(tab, WRITE, SimulatedIterations.getValue('A'), "b"));
```

```
SimulatedIterations simulation = new SimulatedIterations();  
simulation.addTransaction(t1);  
simulation.addTransaction(t2);  
simulation.run(100);
```

Gera uma saída textual mostrando o schedule de execução

SimulatedIterations

- Simula o processamento dos dados nos mesmos moldes de iter

```
Table tab= Utils.createTable("c:\\teste\\it
```

```
Transaction t1 = new Transaction();  
t1.addInstruction(new Instruction(tab, R  
t1.addInstruction(new Instruction(tab, W
```

```
Transaction t2 = new Transaction();  
t2.addInstruction(new Instruction(tab, R  
t2.addInstruction(new Instruction(tab, W
```

```
SimulatedIterations simulation = new Si  
simulation.addTransaction(t1);  
simulation.addTransaction(t2);  
simulation.run(100);
```

Saída gerada:

1 read A

2 read D

4 read F

5 write B

4 read G

4 read A

4 commit

5 write F

5 read G

5 commit

1 write B

1 commit

2 read B

2 write C

2 read H

2 commit

3 write D

3 read E

3 read B

3 commit

Simplificações

- Apenas leituras (READs) e atualizações (WRITEs) são suportadas
 - Operações de exclusão e inserção não foram consideradas
- Transações abortadas não são revertidas
 - São apenas reiniciadas

Simplificações

- O gerenciamento não roda como um serviço separado
 - Tanto o gerenciamento como as transações fazem parte da mesma thread
 - Desse modo, o gerenciador não mantém a transação em modo de espera
 - Em vez disso, devolve nulo para indicar que a transação não pode prosseguir
 - Por causa disso foi necessário um ambiente de simulação que estabelece os momentos em que cada transação é “acordada” no lado gerenciador

OBJETIVO DO TRABALHO

Objetivo do trabalho

- A implementação padrão da classe `ConcurrencyManager` não implementa nenhuma estratégia de prevenção/detecção de deadlock
 - Ou seja, nunca alguma transação é escolhida para ser abortada
- Caso o sistema entre em deadlock
 - as transações envolvidas ficaram esperando por uma liberação que nunca virá

```
public Transaction addToQueue(Item item, Instruction instruction) {  
  
    Transaction t = instruction.getTransaction();  
    Lock l = new Lock(t, instruction.getMode());  
    item.locks.add(l);  
    instruction.setItem(item);  
    return null;  
}
```

Objetivo do trabalho

- O objetivo do trabalho é implementar uma extensão da classe `ConcurrencyManager`
 - a classe deve se chamar `xxxConcurrencyManager`, onde `xxx` é o nome do aluno
- A extensão deve implementar as estratégias de prevenção de deadlock
 - Wait-Die (não preemptiva)
 - Wound-Wait (preemptiva)

Objetivo do trabalho

- A extensão deve
 - implementar um construtor onde a estratégia a ser usada é escolhida
 - Sobrescrever a função addToQueue, que deverá devolver a transação a ser abortada com base na estratégia escolhida

```
public class XXXConcurrencyManager extends ConcurrencyManager{  
  
    public XXXConcurrencyManager(boolean preemptive) throws Exception{  
        ...  
    }  
  
    @Override  
    public Transaction addToQueue(Item item, Transaction t, Instruction instruction) {  
        ....  
    }  
}
```

Exemplo

- A mudança da classe de controle de concorrência é feita dentro do método run da classe SimulatedIterations
- No exemplo abaixo, foi escolhida a estratégia de prevenção não preemptiva Wait-Die

```
public void run(int error) throws Exception {  
  
    //ConcurrencyManager manager = new ConcurrencyManager();  
    ConcurrencyManager manager = new XXXConcurrencyManager(false);  
    int committed = 0;  
    ....  
}
```

Exemplo

- Suponha que as transações abaixo devam ser submetidas ao gerenciador de acesso concorrente

T1: read(B); write(A); write(C).	T2: read(A); read(B); read(C).	T3: write(C); read(A); write(B).
--	--	--

Exemplo

- Trecho de código responsável pelo setup das transações

```
Table table1 = Utils.createTable("c:\\teste\\ibd", "t1.ibd", 1000, true, 1);
```

```
Transaction t1 = new Transaction();
```

```
t1.addInstruction(new Instruction(table1, READ, SimulatedIterations.getValue('B'), null));
```

```
t1.addInstruction(new Instruction(table1, WRITE, SimulatedIterations.getValue('A'), "bla"));
```

```
t1.addInstruction(new Instruction(table1, WRITE, SimulatedIterations.getValue('C'), "bla"));
```

```
Transaction t2 = new Transaction();
```

```
t2.addInstruction(new Instruction(table1, READ, SimulatedIterations.getValue('A'), null));
```

```
t2.addInstruction(new Instruction(table1, READ, SimulatedIterations.getValue('B'), null));
```

```
t2.addInstruction(new Instruction(table1, READ, SimulatedIterations.getValue('C'), null));
```

```
Transaction t3 = new Transaction();
```

```
t3.addInstruction(new Instruction(table1, WRITE, SimulatedIterations.getValue('C'), "bla"));
```

```
t3.addInstruction(new Instruction(table1, READ, SimulatedIterations.getValue('A'), null));
```

```
t3.addInstruction(new Instruction(table1, WRITE, SimulatedIterations.getValue('B'), "bla"));
```


Exemplo

- A chamada ao **run** executa as transações
 - A estratégia de controle de concorrência usada será aquela determinada dentro do método **run**

```
SimulatedIterations simulation = new SimulatedIterations();  
simulation.addTransaction(t1);  
simulation.addTransaction(t2);  
simulation.addTransaction(t3);  
simulation.run(100);
```

Exemplo

- Resultado usando a estratégia padrão, sem nenhuma espécie de controle de deadlock

```
1 read B
    2 read A
        3 write C
    2 read B
```

- O sistema entre em deadlock.

Exemplo

- Resultado usando a estratégia preemptiva Wound-Wait

```
1 read B
    2 read A
        3 write C
    2 Abort
1 write A
    3 Abort
1 write C
1 commit
    2 read A
        3 write C
    2 read B
        3 read A
        3 Abort
    2 read C
    2 commit
        3 write C
        3 read A
        3 write B
        3 commit
```

Exemplo

- Resultado usando a estratégia não preemptiva Wait-Die

1 read B	
2 read A	
3 write C	
2 read B	
3 Abort	
2 read C	
3 Abort	
2 commit	
3 write C	
1 write A	
3 Abort	
1 write C	
3 Abort	
1 commit	
3 write C	
3 read A	
3 write B	
3 commit	

Forma de Verificação

- Consiste em verificar se o schedule correto é produzido
- Crie testes com diferentes transações, variando
 - quantidade de transações
 - quantidade de instruções
 - modo das instruções (read, write)

Entrega

- Prazo final de entrega, sem descontos
 - Sábado, 13 de junho às 23:55
- A cada dia de atraso, a nota é decrementada em 50%.
- O que entregar
 - O código fonte da classe de controle de concorrência (.java) não comprimido