Term Project Proposal(Final)

Chapter 14

Transaction Management

CONTENTS

- **Recovery : Checkpoint**
  - Introduction
  - SimpleDB's checkpoint strategy : *Quiescent checkpoint*
  - Modified checkpoint strategy : *Nonquiescent checkpoint*

- **Concurrency Control : Deadlock**
  - Introduction
  - SimpleDB's deadlock strategy: *Time-limit*
  - Modified deadlock strategy: *Wait-die*

Transactions

- **A transaction is a group of operations that behaves as a single operation.**

- **All transactions should satisfy the ACID properties.**

  - Recovery and concurrency managements are related to the properties.

    - **The *atomicity* and *durability* properties**
      : The proper behavior of the *commit* and *rollback* operations.

    - **The *consistency* and *isolation* properties**
      : The proper behavior of concurrent clients.

Recovery Management

- ***Recovery* is performed each time the database system starts up.**
  - It's purpose is to restore the database to a **reasonable state.**
  - The log contains the history of every modification to the database.

- **The recovery algorithm can stop the as soon as it knows:**
  - All earlier log records were written by completed transactions.
  - The buffer for those transactions have been flushed to disk.

- **Recovery strategies**
  - Quiescent Checkpointing
  - Nonquiescent Checkpointing

Quiescent Checkpointing

- **The recovery algorithm never need to look at the log records prior to a quiescent checkpoint record.**
  - A good time to write a quiescent checkpoint record
    - After recovery has completed
    - Before new transactions have begun

```
<START, 0>
<SETINT, 0, student.tbl, 0, 38, 2004, 2005>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
      //The quiescent checkpoint procedure starts here
<SETSTRING, 0, student.tbl, 0, 46, amy, aimee>
<COMMIT, 0>
      //tx 3 wants to start here, but must wait
<SETINT, 2, junk, 66, 8, 0, 116>
<COMMIT, 2>
<CHECKPOINT>
<START, 3>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Figure 14-10
A log using quiescent checkpointing

Nonquiescent Checkpointing

- **The recovery algorithm never needs to look at the log records prior to the start record of the earliest transaction listed in a nonquiescent checkpoint record.**
  - It simply places the id's of the existing transactions into the checkpoint record.

```
<START, 0>
<SETINT, 0, student.tbl, 0, 38, 2004, 2005>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
<NQCKPT, 0, 2>
<SETSTRING, 0, student.tbl, 0, 46, amy, aimee>
<COMMIT, 0>
<START, 3>
<SETINT, 2, junk, 66, 8, 0, 116>
<SETINT, 3, junk, 33, 8, 543, 120>
```

0, 2 are still running.

Figure 14-12
A log using nonquiescent checkpointing

Nonquiescent Checkpointing

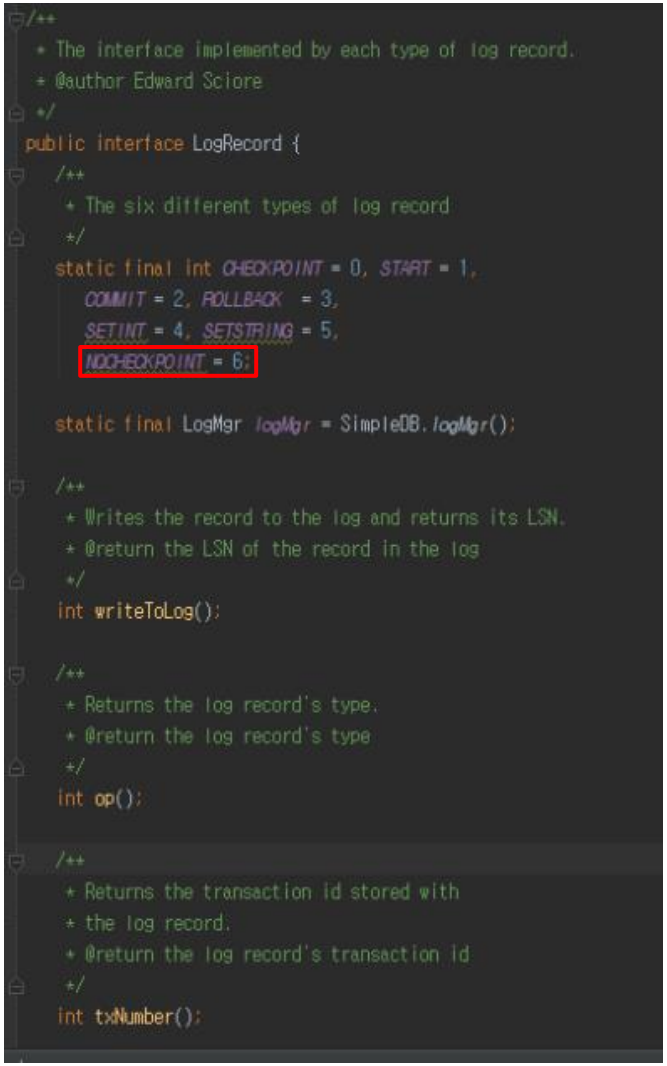
- **The recovery algorithm never needs to look at the log records prior to the start record of the earliest transaction listed in a nonquiescent checkpoint record.**
  - It simply places the id's of the existing transactions into the checkpoint record.

```
<START, 0>
<SETINT, 0, student.tbl, 0, 38, 2004, 2005>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
<NQCKPT, 0, 2>
<SETSTRING, 0, student.tbl, 0, 46, amy, aimee>
<COMMIT, 0>
<START, 3>
<SETINT, 2, junk, 66, 8, 0, 116>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Never needs to look at the log records!

Figure 14-12
A log using nonquiescent checkpointing

Adding non-quiescent checkpointing



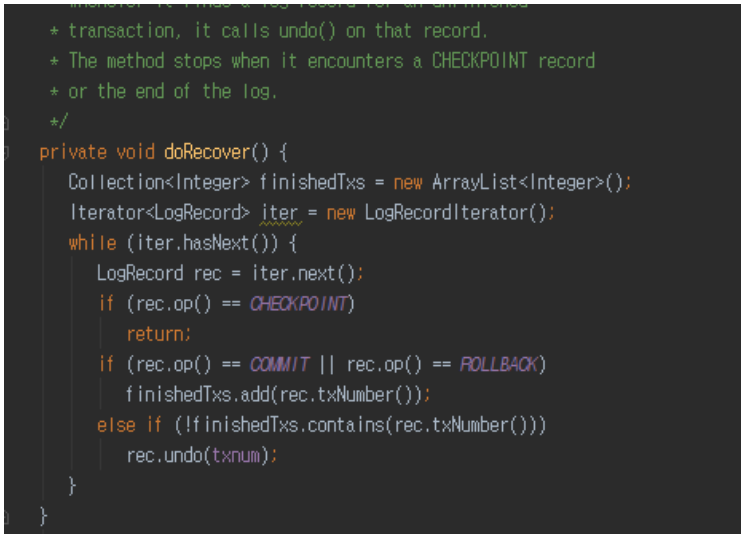
```java
/**
 * The interface implemented by each type of log record.
 * @author Edward Sciore
 */
public interface LogRecord {
   /**
    * The six different types of log record
    */
   static final int CHECKPOINT = 0, START = 1,
      COMMIT = 2, ROLLBACK  = 3,
      SETINT = 4, SETSTRING = 5,
      NOCHECKPOINT = 6;

   static final LogMgr logMgr = SimpleDB.logMgr();

   /**
    * Writes the record to the log and returns its LSN.
    * @return the LSN of the record in the log
    */
   int writeToLog();

   /**
    * Returns the log record's type.
    * @return the log record's type
    */
   int op();

   /**
    * Returns the transaction id stored with
    * the log record.
    * @return the log record's transaction id
    */
   int txNumber();
```

Adding non-quiescent checkpointing


```
class NQCheckpointRecord implements LogRecord {
    private int txnum;
    /**
     * Creates a non-quiescent checkpoint record.
     */
    public NQCheckpointRecord(int txnum) { this.txnum = txnum; }

    /**
     * Creates a log record by reading no other values
     * from the basic log record.
     * @param rec the basic log record
     */
    public NQCheckpointRecord(BasicLogRecord rec) { txnum = rec.nextInt(); }

    /**
     * Writes a non-quiescent checkpoint record to the log.
     * This log record contains the NQCHECKPOINT operator,
     * and nothing else.
     * @return the LSN of the last log value
     */
    public int writeToLog() {
        Object[] rec = new Object[] {NQCHECKPOINT, txnum};
        return logMgr.append(rec);
    }

    public int op() { return NQCHECKPOINT; }

    /**
     * Non-quiescent checkpoint records have no associated transaction,
     * and so the method returns a "dummy", negative txid.
     */
    public int txNumber() { return -1; // dummy value }

    /**
     * Does nothing, because a non-checkpoint record
     * contains no undo information.
     */
```

```
/**
 * Writes a commit record to the log, and flushes it to disk.
 */
public void commit() {
    SimpleDB.bufferMgr().flushAll(txnum);
    int lsn = new CommitRecord(txnum).writeToLog();
    SimpleDB.logMgr().flush(lsn);

    int cTxNum = 0;
    for(int i=0; i<Transaction.currentTxTable.size(); i++) {
        cTxNum = Transaction.currentTxTable.get(i);

        lsn = new NQCheckpointRecord(cTxNum).writeToLog();
        SimpleDB.logMgr().flush(lsn);
    }
}

/**
 * Writes a rollback record to the log, and flushes it to disk.
 */
public void rollback() {
    doRollback();
    SimpleDB.bufferMgr().flushAll(txnum);
    int lsn = new RollbackRecord(txnum).writeToLog();
    SimpleDB.logMgr().flush(lsn);
}

/**
```

Adding non-quiescent checkpointing



```
   * transaction, it calls undo() on that record.
   * The method stops when it encounters a CHECKPOINT record
   * or the end of the log.
   */
  private void doRecover() {
    Collection<Integer> finishedTxs = new ArrayList<Integer>();
    Iterator<LogRecord> iter = new LogRecordIterator();
    while (iter.hasNext()) {
      LogRecord rec = iter.next();
      if (rec.op() == CHECKPOINT)
        return;
      if (rec.op() == COMMIT || rec.op() == ROLLBACK)
        finishedTxs.add(rec.txNumber());
      else if (!finishedTxs.contains(rec.txNumber()))
        rec.undo(txnum);
    }
  }
```

```
private void doRecover() {
 Collection<Integer> finishedTxs = new ArrayList<Integer>();
 ArrayList<Integer> nqcheckedTxs = new ArrayList<Integer>();
 Iterator<LogRecord> iter = new LogRecordIterator();
 while (iter.hasNext()) {
     LogRecord rec = iter.next();
     if (rec.op() == START && nqcheckedTxs.contains(rec.txNumber())
             && nqcheckedTxs.size() == 1)
         return;
     if (rec.op() == COMMIT || rec.op() == ROLLBACK) {
         finishedTxs.add(rec.txNumber());

         for(int indexNum=0; indexNum<nqcheckedTxs.size(); indexNum++) {
             if(rec.txNumber()==nqcheckedTxs.get(indexNum)) {
                 nqcheckedTxs.remove(indexNum);
                 break;
             }
         }
     }
     if (rec.op() == NQCHECKPOINT && !nqcheckedTxs.contains(rec.txNumber())
             && !finishedTxs.contains(rec.txNumber()))
         nqcheckedTxs.add(rec.txNumber());
     else if (!finishedTxs.contains(rec.txNumber()))
       rec.undo(txnum);
 }
}
```

Adding non-quiescent checkpointing

```
    private static synchronized int nextTxNumber() {
        nextTxNum++;
        currentTxTable.add(nextTxNum);
        System.out.println("new transaction: " + nextTxNum);
        return nextTxNum;
    }
}
```

```
    public void commit() {
        for(int indexNum=0; indexNum<currentTxTable.size(); indexNum++) {
            if(txnum==currentTxTable.get(indexNum)) {
                currentTxTable.remove(indexNum);
                break;
            }
        }
        recoveryMgr.commit();
        concurMgr.release();
        myBuffers.unpinAll();

        System.out.println("transaction " + txnum + " committed");
    }

    public void fakecommit() {
//      recoveryMgr.commit();
        concurMgr.release();
        myBuffers.unpinAll();
//      System.out.println("transaction " + txnum + " committed");
    }
```

Adding non-quiescent checkpointing


```
public class NonquiescentTest2 {
    public static void main(String[] args) throws InterruptedException {
        SimpleDB.init("testdb");
        System.out.println("database server 1st_ready");
        System.out.println();

        TestA2 t1 = new TestA2();
        new Thread(t1).start();
        Thread.sleep(2000);

        TestB2 t2 = new TestB2();        new Thread(t2).start();
        TestC2 t3 = new TestC2();        new Thread(t3).start();
        TestD2 t4 = new TestD2();        new Thread(t4).start();
        TestE2 t5 = new TestE2();        new Thread(t5).start();
        Thread.sleep(4000);

        SimpleDB.init("testdb");
        System.out.println("database server 2nd_ready");
        System.out.println();
        Thread.sleep(2000);

        TestA2 t6 = new TestA2();
        new Thread(t6).start();
    }
}
```

```
class TestA2 implements Runnable {
    public void run() {
        Transaction tx = new Transaction();
        Block blk1 = new Block("STUDENT.tbl", 1);
        tx.pin(blk1);

        System.out.println("Tx A: read_1 start");
        int a = tx.getInt(blk1, 1);
        int b = tx.getInt(blk1, 9);
        int c = tx.getInt(blk1, 20);
        int d = tx.getInt(blk1, 40);

        System.out.println("offset1: " + a);
        System.out.println("offset9: " + b);
        System.out.println("offset20: " + c);
        System.out.println("offset40: " + d);
        System.out.println("Tx A: read_1 end");
        tx.commit();
    }
}

class TestB2 implements Runnable {
    public void run() {
        Transaction tx = new Transaction();
        Block blk1 = new Block("STUDENT.tbl", 1);
        tx.pin(blk1);

        System.out.println("Tx B2: write start");
        tx.setInt(blk1, 1, 100);
        System.out.println("Tx B2: write end");
        tx.commit();
    }
}
```

Adding non-quiescent checkpointing

- Normal running

```
new transaction: 1
recovering existing database
transaction 1 committed
database server 1st_ready

new transaction: 2
Tx A: read_1 start
offset1: 10
offset9: 20
offset20: 30
offset40: 40
Tx A: read_1 end
transaction 2 committed

new transaction: 3
Tx C2: write start
Tx C2: write end
transaction 3 committed
new transaction: 4
Tx B2: write start
Tx B2: write end
new transaction: 5
new transaction: 6
Tx E2: write start
Tx D2: write start
transaction 4 committed
Tx D2: write end
transaction 6 committed
Tx E2: write end
transaction 5 committed
```

```
new transaction: 7
recovering existing database
transaction 7 committed
database server 2nd_ready

new transaction: 8
Tx A: read_1 start
offset1: 100
offset9: 200
offset20: 300
offset40: 400
Tx A: read_1 end
transaction 8 committed

Process finished with exit code 0
```

Adding non-quiescent checkpointing

- Abnormal example

```
new transaction: 1
recovering existing database
transaction 1 committed
database server 1st_ready

new transaction: 2
Tx A: read_1 start
offset1: 100
offset9: 20
offset20: 300
offset40: 40
Tx A: read_1 end
transaction 2 committed

new transaction: 3
Tx B2: write start
Tx B2: write end
new transaction: 4
new transaction: 5
new transaction: 6
Tx D2: write start
Tx C2: write start
Tx E2: write start
transaction 3 committed
Tx E2: write end
Tx D2: write end
transaction 4 committed
Tx C2: write end
```

```
new transaction: 7
recovering existing database
transaction 7 committed
database server 2nd_ready

new transaction: 8
Tx A: read_1 start
offset1: 10
offset9: 20
offset20: 30
offset40: 40
Tx A: read_1 end
transaction 8 committed

Process finished with exit code 0
```

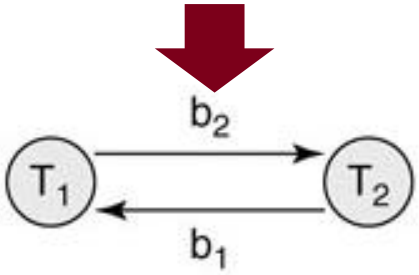
Concurrency Management

- ***Locking***
  - a common technique for ensuring that all schedules are serializable
  - is used to avoid **write-write** and **read-write** conflicts

- **Deadlock**
  - occurs when there is a cycle of transactions
    - The lock protocol does not guarantee that all transactions will commit.
  - exists if the “waits-for” graph has a cycle

- **Deadlock detection strategies**
  - Wait-die
  - Time-limit

$T1 : W(b_1); W(b_2);$
$T2 : W(b_2); W(b_1);$



dead lock example

Time-limit Strategy

- **If a transaction has been waiting for some preset amount of time, the transaction manager can assume that it is deadlocked, and will roll it back.**

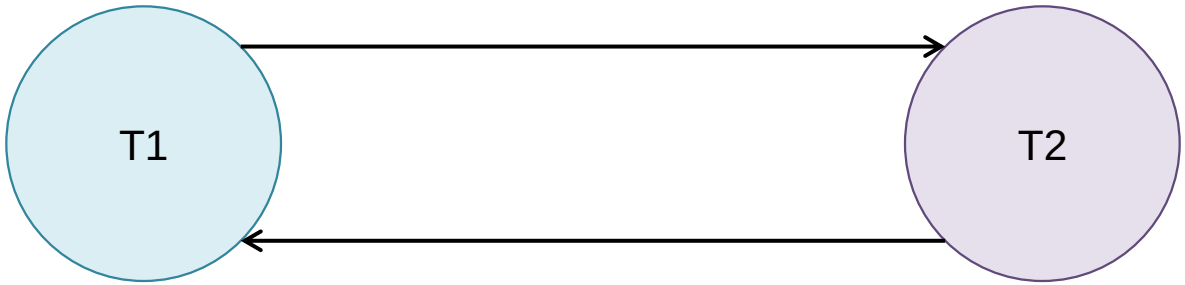
Suppose T_1 requests a lock that conflicts with a lock held by T_2.

1. T_1 waits for the lock.
2. If T_1 stays on the wait list too long then:
   T_1 is rolled back.

Figure 14-23
The time-limit deadlock detection strategy

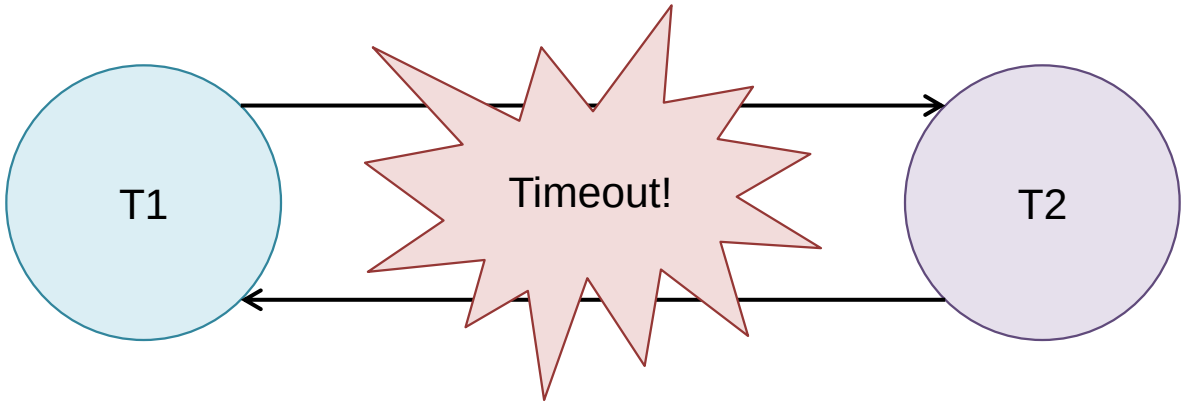
Time-limit Strategy

- **If a transaction has been waiting for some preset amount of time, the transaction manager can assume that it is deadlocked, and will roll it back.**



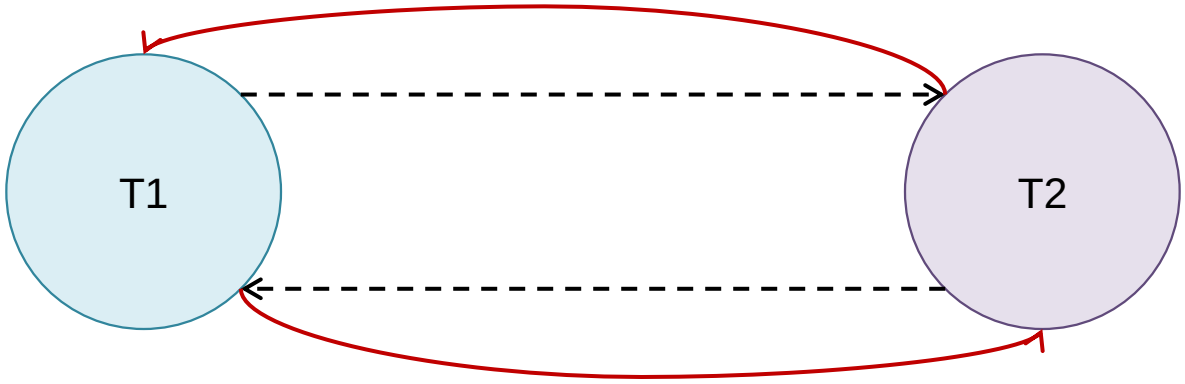
Time-limit Strategy

- **If a transaction has been waiting for some preset amount of time, the transaction manager can assume that it is deadlocked, and will roll it back.**



Time-limit Strategy

- **If a transaction has been waiting for some preset amount of time, the transaction manager can assume that it is deadlocked, and will roll it back.**



Two transactions are rolled back!

Wait-die Strategy

- **This strategy ensures that all deadlocks are detected.**
  - The waits-for graph will contain only edges from older transactions to newer transaction.

Suppose T_1 requests a lock that conflicts with a lock held by T_2.

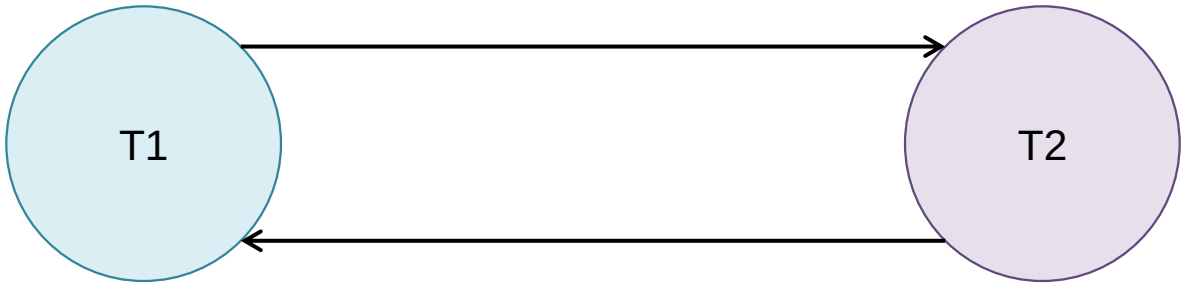
1a. If T_1 is older than T_2, then T_1 waits for the lock.

1b. If T_1 is newer than T_2, then T_1 is rolled back (i.e., it "dies").

Figure 14-22
The wait-die deadlock detection strategy

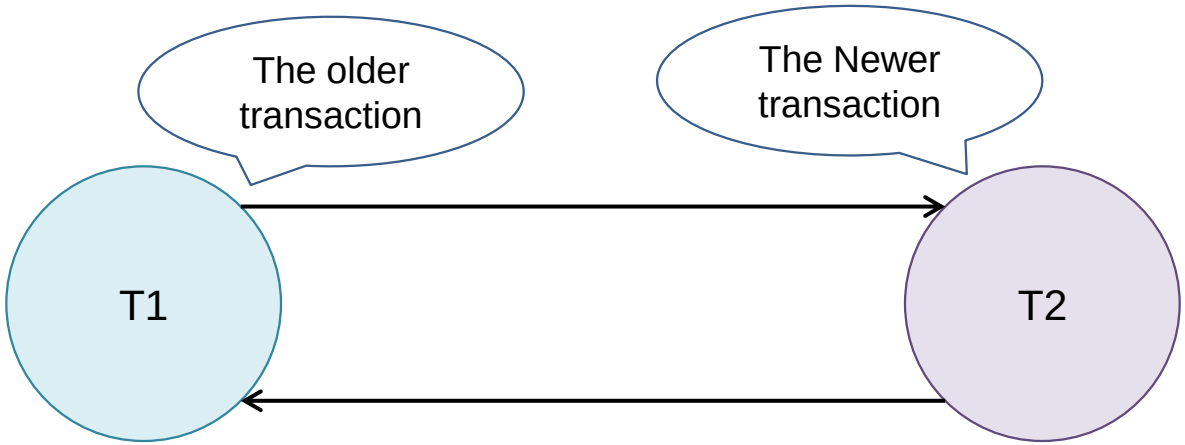
Wait-die Strategy

- **This strategy ensures that all deadlocks are detected.**
  - The waits-for graph will contain only edges from older transactions to newer transaction.



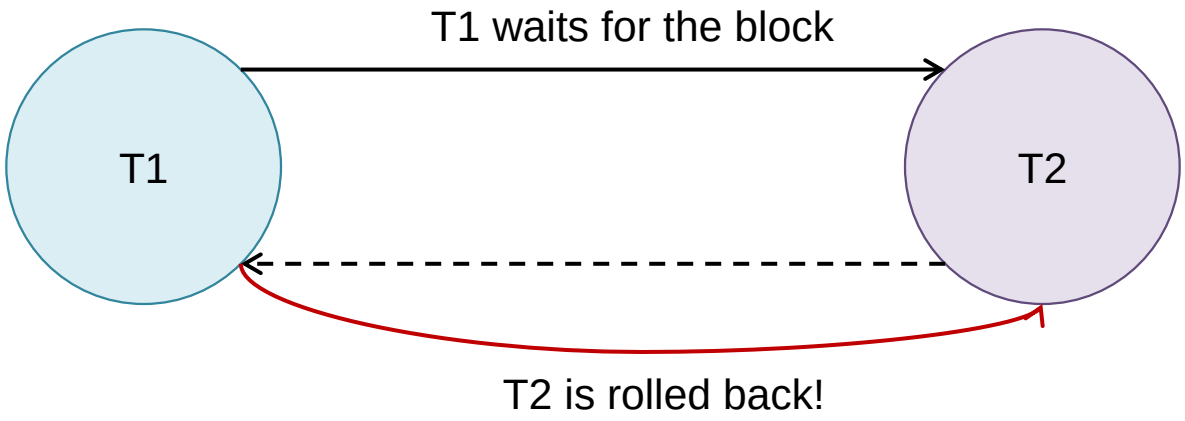
Wait-die Strategy

- **This strategy ensures that all deadlocks are detected.**
  - The waits-for graph will contain only edges from older transactions to newer transaction.



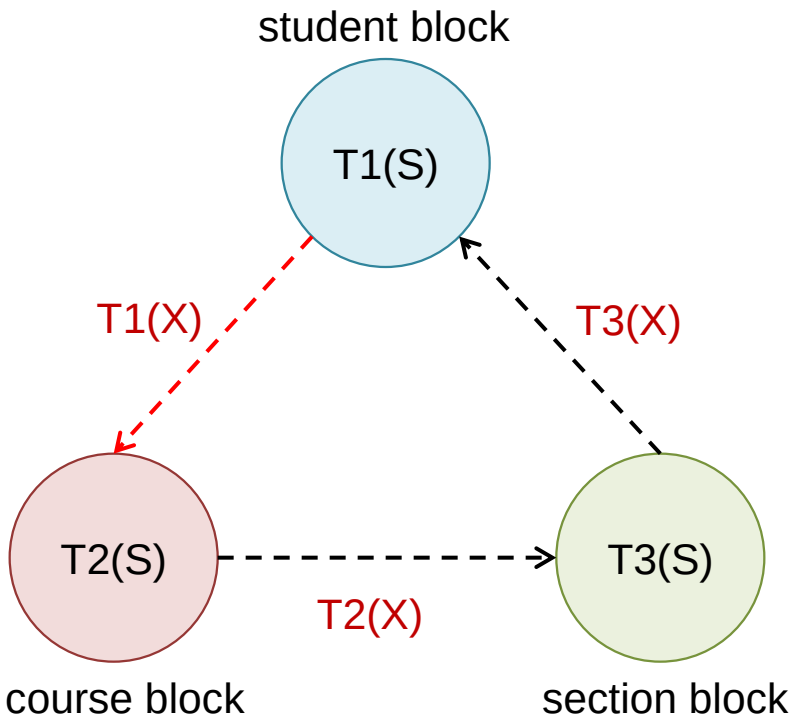
Wait-die Strategy

- **This strategy ensures that all deadlocks are detected.**
  - The waits-for graph will contain only edges from older transactions to newer transaction.



Original SimpleDB : Time-limit

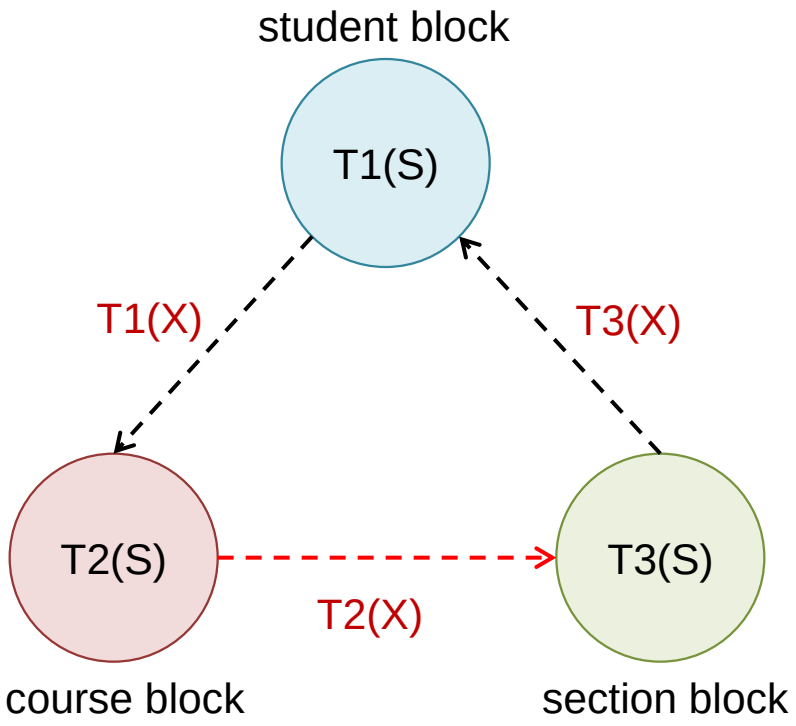
- **Deadlock Example)**




```
package simpledb.server;

import ...

/**
 * Created by shimyeeun on 15. 5. 26.
 */
public class Deadlock_Test {
    public static void main(String args[]) throws Exception {
        // configure and initialize the database
        SimpleDB.init("testdb");

        TestA t1 = new TestA();
        new Thread(t1).start();

        TestB t2 = new TestB();
        new Thread(t2).start();

        TestC t3 = new TestC();
        new Thread(t3).start();
    }


}

class TestA implements Runnable { //read-write
    @Override
    public void run() {
        try {
            Transaction tx = new Transaction();
            int testA;
            Block blk1 = new Block("student.tbl", 1); //(filename, blocknum)
            //Block blk2 = new Block("course.tbl", 1);
            Block blk2 = new Block("section.tbl", 1);

            tx.pin(blk1);
            tx.pin(blk2);

            System.out.println("Tx A: read 1 start");
            testA = tx.getInt(blk1, 4); //(blockname, offset)
            System.out.println("First block of student table is " + testA + ".");
            System.out.println("Tx A: read 1 end");

            Thread.sleep(1000); //1s stop

            System.out.println("Tx A: write 2 start");
            tx.setInt(blk2, 4, 3);
            System.out.println("Tx A: write 2 end");
```

Original SimpleDB : Time-limit

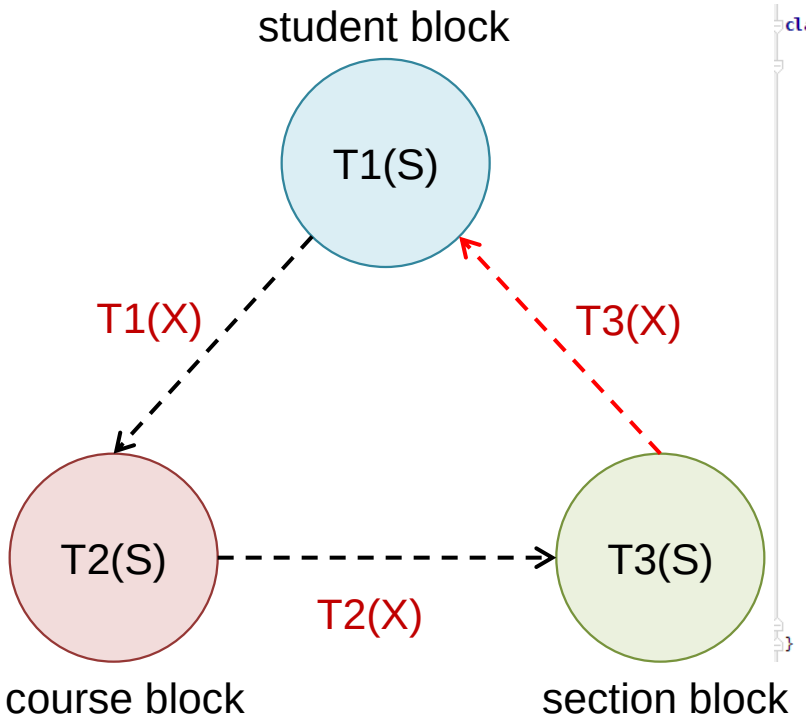
- **Deadlock Example)**




```
            testA = tx.getInt(blk2, 4);
            System.out.println("After write on first block of course table, value is " + testA + ".");
            tx.commit();
        } catch (InterruptedException e) {
        }
        ;
    }
}

class TestB implements Runnable { //write&read
    @Override
    public void run() {
        try {
            Transaction tx = new Transaction();
            int testB;
            Block blk1 = new Block("course.tbl", 1);
            //Block blk2 = new Block("section.tbl", 1);
            Block blk2 = new Block("student.tbl", 1);

            tx.pin(blk1);
            tx.pin(blk2);

            System.out.println("Tx B: read 2 start");
            testB = tx.getInt(blk1, 4); //(blockname, offset, value)
            System.out.println("First block of course table is " + testB + ".");
            System.out.println("Tx B: read 2 end");

            Thread.sleep(1000);

            System.out.println("Tx B: write 3 start");
            tx.setInt(blk2, 4, 5);
            System.out.println("Tx B: write 3 end");

            testB = tx.getInt(blk2, 4);
            System.out.println("After write on first block of section table, value is " + testB + ".");
            tx.commit();
        } catch (InterruptedException e) {
        }
        ;
    }
}

class TestC implements Runnable { //read-write
    @Override
    public void run() {
        try {
            Transaction tx = new Transaction();
            int testC;
```

Original SimpleDB : Time-limit

- **Deadlock Example)**




```
class TestC implements Runnable { //read-write
    @Override
    public void run() {
        try {
            Transaction tx = new Transaction();
            int testC;
            Block blk1 = new Block("section.tbl", 1);
            //Block blk2 = new Block("student.tbl", 1);
            Block blk2 = new Block("course.tbl", 1);

            tx.pin(blk1);
            tx.pin(blk2);

            System.out.println("Tx C: read 3 start");
            testC = tx.getInt(blk1, 4);
            System.out.println("First block of section table is " + testC + ".");
            System.out.println("Tx C: read 3 end");

            Thread.sleep(1000);

            System.out.println("Tx C: write 1 start");
            tx.setInt(blk2, 4, 7);
            System.out.println("Tx C: write 1 end");

            testC = tx.getInt(blk2, 4);
            System.out.println("After write on first block of student table, value is " + testC + ".");
            tx.commit();
        } catch (InterruptedException e) {
        }
        ;
    }
}
```

Original SimpleDB : Time-limit

```
/usr/lib/jvm/java-1.6.0-openjdk-amd64/bin/java ...
new transaction: 1
recovering existing database
transaction 1 committed
new transaction: 2
Tx A: read 1 start
First block of student table is 256.
Tx A: read 1 end
new transaction: 3
new transaction: 4
Tx B: read 2 start
First block of course table is 256.
Tx B: read 2 end
Tx C: read 3 start
First block of section table is 0.
Tx C: read 3 end
Tx A: write 2 start
Tx B: write 3 start
Tx C: write 1 start
```

Wait for the lock

```
/usr/lib/jvm/java-1.6.0-openjdk-amd64/bin/java ...
new transaction: 1
recovering existing database
transaction 1 committed
new transaction: 2
Tx A: read 1 start
First block of student table is 256.
Tx A: read 1 end
new transaction: 3
new transaction: 4
Tx B: read 2 start
First block of course table is 256.
Tx B: read 2 end
Tx C: read 3 start
First block of section table is 0.
Tx C: read 3 end
Tx A: write 2 start
Tx B: write 3 start
Tx C: write 1 start
Exception in thread "Thread-0" simpledb.tx.concurrency.LockAbortException
    at simpledb.tx.concurrency.LockTable.xLock(LockTable.java:107)
    at simpledb.tx.concurrency.ConcurrencyMgr.xLock(ConcurrencyMgr.java:63)
    at simpledb.tx.Transaction.setInt(Transaction.java:189)
    at simpledb.server.TestA.run(Deadlock_Test.java:53)
    at java.lang.Thread.run(Thread.java:701)
Exception in thread "Thread-1" simpledb.tx.concurrency.LockAbortException
    at simpledb.tx.concurrency.LockTable.xLock(LockTable.java:107)
    at simpledb.tx.concurrency.ConcurrencyMgr.xLock(ConcurrencyMgr.java:63)
    at simpledb.tx.Transaction.setInt(Transaction.java:189)
    at simpledb.server.TestB.run(Deadlock_Test.java:86)
    at java.lang.Thread.run(Thread.java:701)
Exception in thread "Thread-2" simpledb.tx.concurrency.LockAbortException
    at simpledb.tx.concurrency.LockTable.xLock(LockTable.java:107)
    at simpledb.tx.concurrency.ConcurrencyMgr.xLock(ConcurrencyMgr.java:63)
    at simpledb.tx.Transaction.setInt(Transaction.java:189)
    at simpledb.server.TestC.run(Deadlock_Test.java:119)
    at java.lang.Thread.run(Thread.java:701)

Process finished with exit code 0
```

After 10s, creates the exception

Original SimpleDB : Time-limit

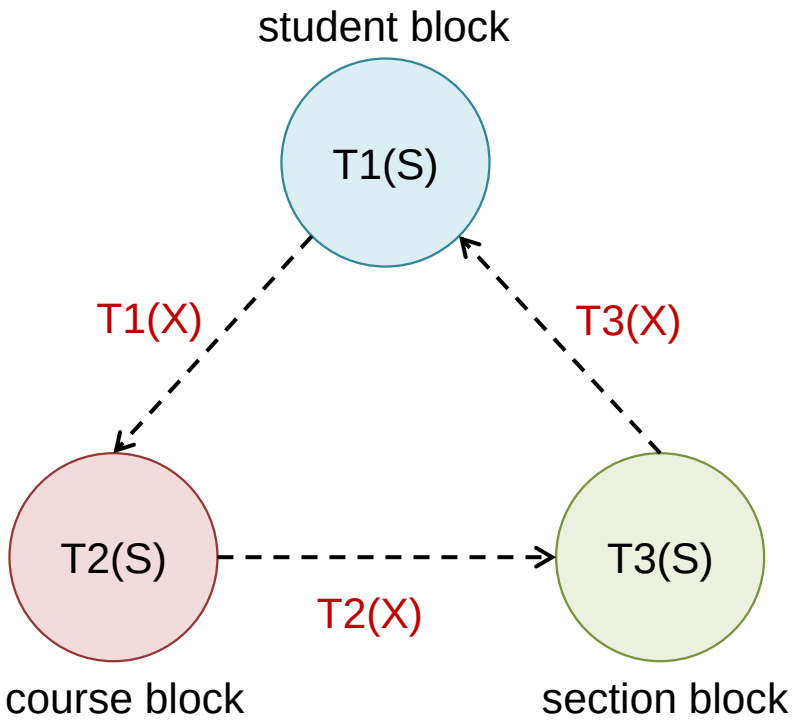
```
public synchronized void sLock(Block blk) { //timeout을 위한 slock method
    try {
        long timestamp = System.currentTimeMillis();

        while (hasXlock(blk) && !waitingTooLong(timestamp))
            wait(MAX_TIME);
        if (hasXlock(blk))
            throw new LockAbortException();

        int val = getLockVal(blk);
        locks.put(blk, val + 1);
    } catch (InterruptedException e) {
        throw new LockAbortException();
    }
}
```

MAX_TIME is 10000(10s)

```
synchronized void xLock(Block blk) {
    try {
        long timestamp = System.currentTimeMillis();
        while (hasOtherSLocks(blk) && !waitingTooLong(timestamp))
            wait(MAX_TIME);
        if (hasOtherSLocks(blk))
            throw new LockAbortException();

        locks.put(blk, -1);

    } catch (InterruptedException e) {
        throw new LockAbortException();
    }
}
```

Modified SimpleDB : Wait-die

- **Deadlock Example)**



Modified SimpleDB : Wait-die

```
/usr/lib/jvm/java-1.6.0-openjdk-amd64/bin/java ...
new transaction: 1
recovering existing database
transaction 1 committed
new transaction: 2
Tx A: read 1 start
First block of student table is 7.
Tx A: read 1 end
new transaction: 3
Tx B: read 2 start
First block of course table is 6.
Tx B: read 2 end
new transaction: 4
Tx C: read 3 start
First block of section table is 0.
Tx C: read 3 end
Tx A: write 2 start
Tx B: write 3 start
Tx C: write 1 start
```

T1, T2 write
the value to block.

```
/usr/lib/jvm/java-1.6.0-openjdk-amd64/bin/java ...
new transaction: 1
recovering existing database
transaction 1 committed
new transaction: 2
Tx A: read 1 start
First block of student table is 7.
Tx A: read 1 end
new transaction: 3
Tx B: read 2 start
First block of course table is 6.
Tx B: read 2 end
new transaction: 4
Tx C: read 3 start
First block of section table is 0.
Tx C: read 3 end
Tx A: write 2 start
Tx B: write 3 start
Tx C: write 1 start
txnum:4, rollback
Exception in thread "Thread-2" java.lang.NullPointerException
    at simpledb.tx.Transaction.getInt(Transaction.java:142)
    at simpledb.server.TestC.run(Deadlock_Test.java:122)
    at java.lang.Thread.run(Thread.java:701)
transaction 4 rolled back
Tx C: write 1 end
Tx B: write 3 end
After write on first block of section table, value is 5.
transaction 3 committed
Tx A: write 2 end
After write on first block of course table, value is 3.
transaction 2 committed

Process finished with exit code 0
```

T3 is rolled back.

Modified SimpleDB : Wait-die(LockTable)

```
*/
class LockTable { //block 단위의 lock
    private static final long MAX_TIME = 10000; // 10 seconds, dead l

    private Map<Block, Integer> locks = new HashMap<~>(); //block과 l
    //-1: xlock, 양수: slock
    private Map<Block, Integer> txnums = new HashMap<~>();
    //현재 txnum 설정
```

```
synchronized int xLock(Block blk, int txnum) { //원래는 void, synchronized int xLock(Block
    try {
        long timestamp = System.currentTimeMillis();
        if(getlockOrder(blk)!=txnum) {
            while ((hasXlock(blk) || hasOtherSLocks(blk)) && !waitingTooLong(timestamp))
                wait(MAX_TIME);
            if (hasOtherSLocks(blk) || hasXlock(blk)) { //만약에 xlock 또는 slock이 존재한다
                //System.out.println("This block has XLOCK or SLOCK. Current txnum of XLO

                if (getlockOrder(blk) < txnum) {
                    return 0;
                }
            }

            if (getlockOrder(blk) > txnum) {
                while (hasOtherSLocks(blk) || hasXlock(blk))
                    wait(1000);
            }
        }
        txnums.put(blk, txnum);
        //System.out.println("xlock settin
        locks.put(blk, -1);
        //System.out.println("Curren lock
        return getLockVal(blk);
    } catch (InterruptedException e) {
        throw new LockAbortException();
    }
}
```

Continue to wait for the lock

```
public synchronized int sLock(Block blk, int txnum) { //다른 s
    try {
        long timestamp = System.currentTimeMillis();

        while (hasXlock(blk) && !waitingTooLong(timestamp)) //
            wait(MAX_TIME); //max time 정도 까지만 wait
        if (hasXlock(blk)) {
            if (getlockOrder(blk) < txnum) {
                return 0; //해당 txnum에 대한 transaction을 rol
            }
        }

        if(getlockOrder(blk) > txnum){
            while (hasXlock(blk)) {
                wait(1000); //계속 대기함. 현재 block의 lock이
            }
        }
        int val = getLockVal(blk);  // will not be negative
        locks.put(blk, val + 1);
        //System.out.println("slock current txnum: " + getloc
        txnums.put(blk, txnum);
        //System.out.println("slock setting txnum: " + getloc
        //System.out.println("Curren lock number: " + getLock
        return getLockVal(blk); //현재 lock을 return
    } catch (InterruptedException e) {
        throw new LockAbortException();
    }
}
```

Modified SimpleDB : Wait-die (Concurrency Mgr)

```
public int sLock(Block blk, int txnum) {
    int result;
    if (locks.get(blk) == null) { //null이 아
        result = locktbl.sLock(blk, txnum); /
        if (result == 0) { //만약에 0이라면 rol
            return 0;
        }
        locks.put(blk, "S");
        return result;
    }
    //S가 존재하거나, 혹은 X가 존재하면 한 transa
    return 1;
}
```

```
public int xLock(Block blk, int txnum) { //원래는
    if (!hasXLock(blk)) { //xlock이 없어야 해! 즉
        int result = locktbl.xLock(blk, txnum);

        if (result == 0)
            return 0;

        locks.put(blk, "X");
        return result;
    }
    //XBlock이 존재할 경우
    return 1;
}
```

Modified SimpleDB : Wait-die (Transaction)


```
public int getInt(Block blk, int offset) {
    int result;
    result = concurMgr.sLock(blk, txnum); //read하는 것이므로 slock을

    if (result == 0) {
        System.out.println("txnum:" + getTxnum() + ", rollback");
        rollback();
    }

    Buffer buff = myBuffers.getBuffer(blk);
    return buff.getInt(offset);
}
```

```
public void setInt(Block blk, int offset, int val) {
    int result;
    result = concurMgr.xLock(blk, getTxnum()); //write하는 것이므로 x

    if (result != 0) { //xlock을 획득한 경우
        Buffer buff = myBuffers.getBuffer(blk);
        int lsn = recoveryMgr.setInt(buff, offset, val);
        buff.setInt(offset, val, txnum, lsn);
    } else { //0을 return 받아 rollback을 해야하는 경우
        System.out.println("txnum:" + getTxnum() + ", rollback");
        rollback();
    }
}
```

Q & A