CS162
Operating Systems and Systems Programming
Lecture 18

Transactions

April 4, 2011

Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Transactions, concurrency control
- Two-phase lock
- Strict two-phase lock

Note: Some slides and/or pictures in the following are adapted from lecture notes by Mike Franklin.

Recap: Read/Writer Example

```
Reader() {
    // check into system
    lock.Acquire();
    while ((AW + WW) > 0) {
      WR++;
      okToRead.wait(&lock);
      WR--;
    }
    AR++;
    lock.release();

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okToWrite.signal();
    lock.Release();
  }
```

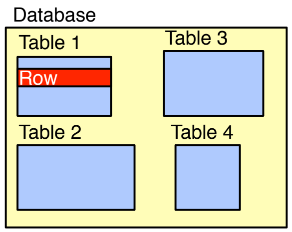
```
Writer() {
    // check into system
    lock.Acquire();
    while ((AW + AR) > 0) {
      WW++;
      okToWrite.wait(&lock);
      WW--;
    }
    AW++;
    lock.release();

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    lock.Acquire();
    AW--;
    if (WW > 0){
      okToWrite.signal();
    } else if (WR > 0) {
      okToRead.broadcast();
    }
    lock.Release();
  }
```


Recap: Read/Writer Example

- Properties:
  - Allow multiple concurrent active readers if no active writer
  - Only one writer at a time
  - If a writer waits, no new active readers are allowed
- Locking granularity: entire database

Locking Granularity

- What granularity to lock?
  - Database
  - Tables
  - Rows



- Fine granularity (e.g., row) → high concurrency
  - Multiple users can update the database and same table simultaneously
- Coarse granularity (e.g., database, table) → simple, but low concurrency

From Multiprogramming to Transactions

- Users would like the illusion of running their programs on the machine alone
  - Why not running the entire program in a critical section?
- Users want fast response time and operators want to increase machine utilization → increase concurrency
  - Interleave executions of multiple programs
- How can DBMS (database management system) help?

Concurrent Execution & Transactions

- Concurrent execution essential for good performance
  - Disk slow, so need to keep the CPU busy by working on several user programs concurrently
- DBMS only concerned about what data is read/written from/ to the database
  - Not concerned about other operations performed by program on data
- **Transaction** - DBMS's abstract view of a user program, i.e., a sequence of reads and writes.

Transaction - Example

```
BEGIN;     --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance -
   100.00 WHERE name = 'Alice';

 UPDATE branches SET balance = balance -
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Alice');

 UPDATE accounts SET balance = balance +
   100.00 WHERE name = 'Bob';

 UPDATE branches SET balance = balance +
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Bob');

COMMIT;     --COMMIT WORK
```

The ACID properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** if each transaction is consistent, and the DB starts consistent, it ends up consistent
- **Isolation:** execution of one transaction is isolated from that of all others
- **Durability:** if a transaction commits, its effects persist

Atomicity

- A transaction
  – might *commit* after completing all its operations, or
  – it could *abort* (or be aborted by the DBMS) after executing some operations
- Atomic Transactions: a user can think of a transaction as always either *executing all its* operations, or *not executing any* operations at all
  – DBMS *logs* all actions so that it can *undo* the actions of aborted transactions

Consistency

- Data in DBMS is accurate in modeling real world, follows integrity constraints (ICs)
- If DBMS is consistent before transaction, it will be after
- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted)
  – DBMS enforces some ICs, depending on the ICs declared in CREATE TABLE statements
  – Beyond this, DBMS does not understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed)

Isolation

- Each transaction executes as if it was running by itself
  – Concurrency is achieved by DBMS, which interleaves operations (reads/writes of DB objects) of various transactions
- Techniques:
  – Pessimistic – don't let problems arise in the first place
  – Optimistic – assume conflicts are rare, deal with them *after* they happen.

Durability

- Data should survive in the presence of
  - System crash
  - Disk crash → need backups
- All committed updates and only those updates are reflected in the database
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

This Lecture

- Deal with **(I)solation**, by focusing on **concurrency control**
- For (A)tomicity, (C)onsistency, and (D)urability take cs186!

Example

- Consider two transactions:
  - T1: moves $100 from account A to account B

```
T1:A := A-100; B := B+100;
```

  - T2: moves $50 from account B to account A

```
T2:A := A+50;  B := B-50;
```

- Each operation consists of (1) a read, (2) an addition/ subtraction, and (3) a write
- Example: A = A-100

```
Read(A); // R(A)
A := A - 100;
Write(A); // W(A)
```


Example (cont'd)

- Database only sees reads and writes

		Database View
`T1: A:=A-100; B:=B+100;`	→	`T1:R(A),W(A),R(B),W(B)`
`T2: A:=A+50; B:=B-50;`	→	`T2:R(A),W(A),R(B),W(B)`

- Assume initially: A = $1000 and B = $500
- What is the legal outcome of running T1 and T2?
  - A = $950
  - B = $550

Example (cont'd)

- What is the outcome of the following execution?

```
T1:R(A),W(A),R(B),W(B)
T2:                     R(A),W(A),R(B),W(B)
```

A=900 B=600 A=950 B=550

- Answer: A = $950, B = $550
- What is the outcome of the following execution?

```
T1:                     R(A),W(A),R(B),W(B)
T2:R(A),W(A),R(B),W(B)
```

A=950 B=550 A=1050 B=450

- Answer: A = $950, B = $550

Example (cont'd)

- What is the outcome of the following execution?

```
T1:R(A),W(A),                       R(B),W(B)
T2:          R(A),W(A),R(B),W(B)
```

A=900 B=550 A=950 B=450

- Answer: A = $950, B = $550
- What is the outcome of the following execution?

```
T1:R(A),                       W(A),R(B),W(B)
T2:     R(A),W(A),R(B),W(B)
```

A=900 B=550 A=1050 B=450

- Answer: A = $900, B = $550; lost $50 !!

Transaction Scheduling

- Why not run only one transaction at a time?
- Answer: low system utilization
  - Two transactions cannot run simultaneously even if they access different data
- Goal of transaction scheduling:
  - Maximize system utilization, i.e., concurency
    - » Interleave operations from different transactions
  - Preserve transaction semantics
    - » Logically the sequence of all operations in a transaction are executed atomically
    - » Intermediate state of a transaction is not visible to other transactions

Transaction Scheduling

- **Serial schedule:** A schedule that does not interleave the operations of different transactions
  - Transactions run serially (one at a time)
- **Equivalent schedules:** For any database state, the effect (on the database) and output of executing the first schedule is identical to the effect of executing the second schedule
- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time.

Anomalies with Interleaved Execution

- May violate transaction semantics, e.g., some data read by the transaction changes before committing
- Inconsistent database state, e.g., some updates are lost
- Anomalies always involves a “write”; Why?

Anomalies with Interleaved Execution

- Read-Write conflict (Unrepeatable reads)

```
T1:R(A),           R(A),W(A)
T2:      R(A),W(A)
```

- Violates transaction semantics
- Example: Mary and John want to buy a TV set on Amazon but there is only one left in stock
  - (T1) John logs first, but waits…
  - (T2) Mary logs second and buys the TV set right away
  - (T1) John decides to buy, but it is too late…

Anomalies with Interleaved Execution

- Write-read conflict (reading uncommitted data)

```
T1:R(A),W(A),           W(A)
T2:            R(A),W(A)
```

- Example:
  - (T1) A user updates value of A in two steps
  - (T2) Another user reads the intermediate value of A, which can be inconsistent
  - Violates transaction semantics since T2 is not supposed to see intermediate state of T1

Anomalies with Interleaved Execution

- Write-write conflict (overwriting uncommitted data)

```
T1:W(A),           W(B)
T2:      W(A),W(B)
```

- Get T1’s update of B and T2’s update of A
- Violates transaction serializability
- If transactions were serial, you’d get either:
  - T1’s updates of A and B
  - T2’s updates of A and B

Conflict Serializable Schedules

- Two operations **conflict** if they
  - Belong to different transactions
  - Are on the same data
  - At least one of them is a write.
- Two schedules are **conflict equivalent** iff:
  - Involve same operations of same transactions
  - Every pair of **conflicting** operations is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

Conflict Equivalence – Intuition

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Example:

```
T1:R(A),W(A),          R(B),W(B)
T2:          R(A),W(A),          R(B),W(B)
```

```
T1:R(A),W(A),     R(B),     W(B)
T2:          R(A),     W(A),     R(B),W(B)
```

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),     R(B),W(B)
```


Conflict Equivalence – Intuition (cont'd)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Example:

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),     R(B),W(B)
```

```
T1:R(A),W(A),R(B),     W(B)
T2:               R(A),     W(A),R(B),W(B)
```

```
T1:R(A),W(A),R(B),W(B)
T2:                   R(A), W(A),R(B),W(B)
```


Conflict Equivalence – Intuition (cont'd)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Is this schedule serializable?

```
T1:R(A),           W(A)
T2:     R(A),W(A),
```

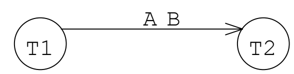
Dependency Graph

- **Dependency graph:**
  - Transactions represented as nodes
  - Edge from Ti to Tj:
    - » an operation of Ti conflicts with an operation of Tj
    - » Ti appears earlier than Tj in the schedule
- **Theorem:** Schedule is conflict serializable if and only if its dependency graph is acyclic

Example

- Conflict serializable schedule:

```
T1:R(A),W(A),             R(B),W(B)
T2:          R(A),W(A),             R(B),W(B)
```



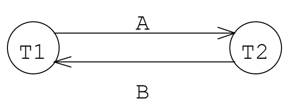
Dependency graph

- No cycle!

Example

- Conflict that is *not* serializable:

```
T1:R(A),W(A),                       R(B),W(B)
T2:          R(A),W(A),R(B),W(B)
```



Dependency graph

- Cycle: The output of T1 depends on T2, and vice-versa

Notes on Conflict Serializability

- Conflict Serializability doesn't allow all schedules that you would consider correct
  - This is because it is strictly *syntactic* - it doesn't consider the meanings of the operations or the data
- In practice, Conflict Serializability is what gets used, because it can be done efficiently
  - Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, …
- Two-phase locking (2PL) is how we implement it

5min Break

Locks

- "Locks" to control access to data
- Two types of locks:
  - shared (S) lock – multiple concurrent transactions allowed to operate on data
  - exclusive (X) lock – only one transaction can operate on data at a time

Lock Compatibility Matrix

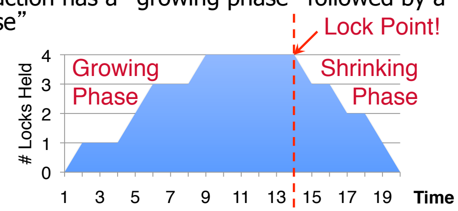
	S	X
S	√	–
X	–	–

Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - S (*shared*) or X (exclusive) lock on data before reading,
   - X (*exclusive*) lock on data before writing

2) A transaction can not request additional locks once it releases any locks.

Thus, each transaction has a "growing phase" followed by a "shrinking phase"



Two-Phase Locking (2PL)

- 2PL guarantees conflict serializability
- Doesn't allow dependency cycles; Why?
- Answer: a cyclic dependency cycle leads to deadlock
  - Edge from Ti to Tj means that Ti acquires lock first and Tj needs to wait
  - Edge from Ti to Tj means that Ti acquires lock first and Tj needs to wait
  - Thus, both T1 and Tj wait for each other → deadlock
- Schedule of conflicting transactions is conflict equivalent to a serial schedule ordered by "lock point"

Lock Management

- Lock Manager (LM) handles all lock and unlock requests
  - LM contains an entry for each currently held lock
- Lock table entry:
  - Pointer to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to **queue of lock requests**
- When lock request arrives see if anyone else holds a conflicting lock
  - If not, create an entry and grant the lock
  - Else, put the requestor on the wait queue
- Locking and unlocking are atomic operations
- Lock upgrade: shared lock can be upgraded to exclusive lock

Deadlock

- Cycles of transactions waiting for each other to release locks
- Recall: two ways to deal with deadlocks
  - Deadlock detection
  - Deadlock prevention
- Many systems punt problem by using timeouts instead
  - Associate a timeout with each lock
  - If timeout expires release the lock
  - What is the problem with this solution?

Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: If Ti is older, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti is older, Tj aborts; otherwise Ti waits
- If a transaction re-starts, make sure it gets its original timestamp
  - Why?

Example

- T1 transfers $50 from account A to account B

```
T1:Read(A),A:=A-50,Write(A),Read(B),B:=B+50,Write(B)
```

- T2 outputs the total of accounts A and B

```
T2:Read(A),Read(B),PRINT(A+B)
```

- Initially, A = $1000 and B = $2000
- What are the possible output values?

Is this a 2PL Schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	<granted>
	Read(A)
	Unlock(A)
	Lock_S(B) <granted>
Lock_X(B)	
	Read(B)
<granted>	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

No, and it is not serializable

Is this a 2PL Schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	<granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Yes, so it is serializable

Cascading Aborts

- Example: T1 aborts
  - Note: this is a 2PL schedule

```
T1:R(A),W(A),           R(B),W(B), Abort
T2:               R(A),W(A)
```

- Rollback of T1 requires rollback of T2, since T2 reads a value written by T1
- Solution: **Strict Two-phase Locking (Strict 2PL)**: same as 2PL except
  - All locks held by a transaction are released only when the transaction completes

Strict 2PL (cont'd)

- All locks held by a transaction are released only when the transaction completes
- In effect, "shrinking phase" is delayed until:
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (then locks can be released after rollback).

Is this a Strict 2PL schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	<granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

No: Cascading Abort Possible

Is this a Strict 2PL schedule?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	<granted>
	Read(A)
	Lock_S(B) <granted>
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

Summary

- Correctness criterion for transactions is "serializability".
  - In practice, we use "conflict serializability", which is somewhat more restrictive but easy to enforce.
- Two Phase Locking, and Strict 2PL: Locks directly implement the notions of conflict
  - The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Much more about transactions in cs186