6.033: Fault Tolerance: Isolation
Lecture 17
Katrina LaCurts, lacurts@mit.edu

0. Introduction
   - Last time: Atomicity via logging. We're good with atomicity now.
   - Today: Isolation
   - The problem: We have multiple transactions -- T1, T2, .., TN -- all of which must be atomic, and all of which can have multiple steps. We want to schedule the steps of these transactions so that it appears as if they ran sequentially.
   - Naive solution: run transaction sequentially with a single global lock.
     - Very poor performance
   - Better solution: fine-grained locking. But we all agreed that this was error prone back in the OS section. What to do?

1. Serializability
   - What does it mean for transactions to "appear" as if they were run in sequence?
     - That the final written state is the same?
     - That the final written state + intermediate reads are the same?
     - Something else?
   - It depends! There are different types of "serializability". The right one depends on what your application is doing.
   - Final-state serializability: A schedule is final-state serializable if its final written state is equivalent to that of some sequential schedule.
     - We won't spend much time on this

2. Conflict Serializability
   - Two operations conflict if
     1. They both operate on the same object
     2. At least one of them is a write
   - Definition highlights that concurrent reads are generally fine, but problems arise as soon as writes get involved.
   - A schedule is conflict serializable if the order of all of its conflicts is the same as the order of the conflicts in some sequential schedule.
     - By "order of conflicts" we mean the ordering of the steps in each individual conflict.
   - See slides for examples. A schedule can be final-state serializable but not conflict serializable.

3. Conflict graphs
   - Nodes are transactions
   - Edges are directed
   - There is an edge from T_i to T_j iff
     - T_i and T_j have a conflict between them
     - The first step in the conflict occurs in T_i

- Example 1:

```
T2: write(x,20)
T1: read(x)
T2: write(y,30)
T1: read(y)
T1: write(y, y+10)
```

  There are three conflicts:

```
T2: write(x,20); T1: read(x)
T2: write(y,30); T1: read(y)
T2: write(y,30); T1: write(y,y+10)
```

  In each conflict, the first step in the schedule is in T2.
  Conflict graph is: T2 -> T1.

- Example 2:

```
T1: read(x)
T2: write(x,20)
T2: write(y,30)
T1: read(y)
T1: write(y, y+10)
```

  Now our three conflicts are:

```
T1: read(x); T2: write(x,20)
T2: write(y,30); T1: read(y)
T2: write(y,30); T1: write(y,y+10)
```

  Our conflict graph here is T1 <--> T2

  (Note: this schedule was final-state serializable but not conflict serializable)

- Example 3: (not covered in lecture; this is good practice!)

```
T1: read(x)
T2: write(x)
T3: read(y)
T4: read(y)
T1: write(y)
T2: write(y)
T3: write(z)
```

  The conflicts here are:

```
T1: read(x); T2: write(x)
T3: read(y); T1: write(y)
T3: read(y); T2: write(y)
T4: read(y); T1: write(y)
T4: read(y); T2: write(y)
```

```
T1: write(y); T2: write(y)
```

The conflict graph is:

```
       |----------<--
       v            |
T1 --> T2 <-- T3    T4
^^            |     |
||------<-----|     |
|                   |
|----------<--------|
```

- Acyclic conflict graph <=> conflict-serializable
  - Intuition: conflict graph for any sequential schedule is acyclic
  - (But we won't formally prove this)

4. Interlude
   - We're going to explore conflict serializability in more depth because it's useful in practice.
   - As of right now, we have no methodical way to create conflict serializable schedules
     - We can check if a schedule is conflict serializable, but if we want a conflict serializable schedule the best we can do right now is keep generating random schedules and testing until we find one with an acyclic conflict graph

5. Two-phase locking (2PL)
   - So how do we generate conflict-serializable schedules?
   - Via two-phase locking:
     - Each shared variable has a lock (fine-grained locking).
     - Before any operation on the variable, the transaction must acquire the corresponding lock.
     - After a transaction releases a lock, it may not acquire any other locks.
   - (Proof that 2PL => conflict serializability is at the end)
   - Note: fine-grained locking but in a systematic way
   - Why "two-phase"? We see two phases:
     - Acquire phase, where transactions acquire locks
     - Release phase, where transactions release locks
   - Immediate problem: 2PL can result in deadlock:

```
T1                  T2
acquire(x_lock)     acquire(y_lock)
read(x)             read(y)
acquire(y_lock)     acquire(x_lock)
read(y)             read(x)
release(y_lock)     release(x_lock)
release(x_lock)     release(y_lock)
```

- One solution to deadlock: global ordering on locks. Not very modular.
- Better solution: take advantage of atomicity and abort one of the transactions.
  - Seems like we're punting, but is actually very elegant; given atomicity, aborting is a-okay.
  - Detecting deadlock is possible
    - Use "wait-dependency" graphs, which capture the locks each transaction has and the ones it wants. Cycle in wait-dependency graph => deadlock
    - Or just abort a transaction after X seconds (X "reasonably" large). Not as elegant, but simpler.

6. Performance Improvement: Reader-writer locks
   - Reader-writer locks.
   - Rules:
     - Can acquire a reader lock at the same time as other readers.
     - Can acquire a writer lock only if there are no other readers or writers.
   - What about fairness? If readers keep acquiring the lock, and a writer is waiting?
     - Typically: if writer is waiting, new readers wait too.
   - Reader-writer locks improve performance
     - As described, they allow for concurrent reads.
     - We can also release read locks prior to commit
       - Why? Once a transaction T has acquired all its locks (reached its "lock point") any conflict transaction will run later
       - If T reaches its lock point and will no longer access X, releasing read locks on X will be fine
       - Hold write locks until commit, though, in case the transaction aborts.
   - Can also improve performance by relaxing our requirements for serializability/isolation.
     - Read-committed or snapshot isolation (see future hands-on)
   - Important: there are a ton of tradeoffs between performance and isolation semantics.

7. Another Possible Performance improvement: Giving up on Conflict Serializability
   - Sometimes conflict serializability can seem like too strict a requirement
   - Example:

```
T1        T2        T3
read(x)
          write(x)
write(x)
                    write(x)
```

Conflict graph:

```
T1 <--> T2
 |      |
 v      |
 T3 <----
```

- Not acyclic => not conflict serializable
- But compare it to running T1, then T2, then T3 (sequentially)
  - Final-state is fine
  - Intermediate reads are fine
- So what's wrong? Why shouldn't we allow this schedule?
- This schedule is view serializable, but not conflict serializable
  - Informally: a schedule is view serializable if the final written state as well as intermediate reads are the same as in some sequential schedule.
  - Formally, for those interested (this will NOT be on any exam)
    - Two schedules S and S' are view equivalent if:
      - If T_i in S reads an initial value for X, so does T_i in S'
      - If T_i in S reads the value written by T_j in S for some X, so does T_i in S'
      - If T_i in S does the final write to X, so does T_i in S'
    - A schedule is view serializable if it is view equivalent to some sequential schedule.
- Why focus on conflict serializability when it seems too strict? Why not focus on view serializability?
  - View serializability is hard to test for. Conflict serializability is not, since checking whether a graph is acyclic is fast.
  - We have an easy way to generate conflict serializable schedules.
    - Conflict serializable schedules are also view serializable, so technically this means we have an easy way to generate view serializable schedules. But we don't have an easy way to generate view schedules that allows for ones like the example above.
  - Schedules that are view serializable but not conflict serializable involve blind writes: writes that are ultimately not read. These are not common in practice.
- Basically: conflict serializability has practical benefits

8. Summary
   - Now: Have atomicity and isolation working on a single machine (concurrent transactions, good performance, etc.)
   - Next week: distributed transactions

Proof that 2PL produces a conflict-serializable schedule. We would never expect you to reproduce this proof, or any similar proof, on an exam.

(i) Suppose not. Suppose the conflict graph produced by an execution of 2PL has a cycle, which without loss of generality, is T1 --> T2 --> ... --> Tk --> T1.

(ii) We'll show that a locking protocol that produces such a schedule must violate 2PL.

(iii) Let the shared variable -- the one that causes the conflict -- between T_i and T_{i+1} be represented by x_i.

T1 and T2 conflict on x1
T2 and T3 conflict on x2
...
Tk and T1 conflict on x_k

(iv) This means that:

T1 acquires x1.lock
T2 acquires x1.lock and x2.lock
T3 acquires x2.lock and x3.lock
...
Tk acquires x_k.lock and x_{k-1}.lock
T1 acquires x_k.lock

(v) Time flows down in the above step. Since the edges go from T_i to T_{i+1}, T_i must have accessed x_i before T_{i+1}.

(vi) For T2 to have acquired its locks -- in particular, x1.lock -- T1 must have previously released x1.lock. Thus:

T1 acquires x1.lock
T1 releases x1.lock
T2 acquires x1.lock and x2.lock
T3 acquires x2.lock and x3.lock
...
Tk acquires x_k.lock and x_{k-1}.lock
T1 acquires x_k.lock

(vii) Focusing just on the steps that involve T1:
T1 acquires x1.lock
T1 releases x1.lock
T1 acquires x_k.lock

(viii) T1 violates 2PL; it acquires a lock after releasing a lock

(ix) Therefore, cyclic conflict graph => 2PL was violated.
Alternatively, 2PL => acyclic conflict graph