6.033 Spring 2017

Lecture #17

- **Isolation**
  - **Conflict serializability**
  - **Conflict graphs**
  - **Two-phase locking**

goal: build reliable systems from unreliable components

the abstraction that makes that easier is

transactions, which provide **atomicity** and **isolation**, while not hindering **performance**

atomicity → **shadow copies** (simple, poor performance) or **logs** (better performance, a bit more complex)

isolation → **two-phase locking**

eventually, we also want transaction-based systems to be **distributed**: to run across multiple machines

goal: run transactions `T1`, `T2`, .., `TN` concurrently, and have it "appear" as if they ran sequentially

```
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit
```

```
T2
begin
write(x, 20)
write(y, 30)
commit
```

naive approach: actually run them sequentially, via (perhaps) a single global lock

goal: run transactions `T1`, `T2`, .., `TN` concurrently, and have it "appear" as if they ran sequentially

↑ **what does this even mean?**

```
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit
```

```
T2
begin
write(x, 20)
write(y, 30)
commit
```

```
T1                       T2
begin                    begin
read(x)                  write(x, 20)
tmp = read(y)            write(y, 30)
write(y, tmp+10)         commit
commit
```

possible sequential schedules

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

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

at end:

x=20, y=40

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

at end:

x=20, y=10

(assume x, y initialized to zero)

```
T1                       T2
begin                    begin
read(x)                  write(x, 20)
tmp = read(y)            write(y, 30)
write(y, tmp+10)         commit
commit
```

possible sequential schedules

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

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

at end:

x=20, y=40

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

at end:

x=20, y=10

(assume x, y initialized to zero)

```
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit
```

```
T2
begin
write(x, 20)
write(y, 30)
commit
```

possible sequential schedules

T1 -> T2: x=20, y=30

T2 -> T1: x=20, y=40

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

at end:
x=20, y=40

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

at end:
x=20, y=40

```
T1                      T2
begin                   begin
read(x)                 write(x, 20)
tmp = read(y)           write(y, 30)
write(y, tmp+10)        commit
commit
```

possible sequential schedules

`T1 -> T2: x=20, y=30`
`T2 -> T1: x=20, y=40`

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

at end:
`x=20, y=40`

```
T1: read(x) // x=0
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y) // y=30
T1: write(y, tmp+10)
```

at end:
`x=20, y=40`

In the second schedule, `T1` reads `x=0` *and* `y=30`; those two reads together aren't possible in a sequential schedule. is that okay?

it depends.

there are many ways for multiple transactions to "appear" to have been run in sequence; we say there are different notions of **serializability**. what type of serializability you want depends on what your application needs.

conflicts

two operations conflict if they operate on the same object and at least one of them is a write.

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

conflicts

`T1.1 read(x)` and `T2.1 write(x, 20)`
`T1.2 tmp = read(y)` and `T2.2 write(y, 30)`
`T1.3 write(y, tmp+10)` and `T2.2 write(y, 30)`

conflicts

two operations conflict if they operate on the same object and at least one of them is a write.

in any schedule, two conflicting operations A and B will have an order: either A is executed before B, or B is executed before A. we'll call this the **order** of the conflict (in that schedule).

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

---

conflicts

```
T1.1 read(x)            ->  T2.1 write(x, 20)
T1.2 tmp = read(y)      ->  T2.2 write(y, 30)
T1.3 write(y, tmp+10)   ->  T2.2 write(y, 30)
```

if we execute T1 before T2, within any conflict, T1's operation will occur first

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

---

conflicts

```
        T1.1 read(x)   <-  T2.1 write(x, 20)
   T1.2 tmp = read(y)  <-  T2.2 write(y, 30)
T1.3 write(y, tmp+10)  <-  T2.2 write(y, 30)
```

if we execute **T2** before **T1**, within any conflict, **T2**'s operation will occur first

conflicts

two operations conflict if they operate on the same object and at least one of them is a write.

conflict serializability

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.

conflicts

T1.1, T2.1
T1.2, T2.2
T1.3, T2.2

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.

(here, that means we will see one transaction's — T1's or T2's — operation occurring first in each conflict)

```
T2.1: write(x, 20)
T1.1: read(x)
T2.2: write(y, 30)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)
```

T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3

```
T1.1: read(x)
T2.1: write(x, 20)
T2.2: write(y, 30)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)
```

T1.1 -> T2.1
T2.2 -> T1.2
T2.2 -> T1.3

conflicts

T1.1, T2.1
T1.2, T2.2
T1.3, T2.2

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.

(here, that means we will see one transaction's — T1's or T2's — operation occurring first in each conflict)

```
T2.1: write(x, 20)
T1.1: read(x)
T2.2: write(y, 30)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)
```

T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3

~~```
T1.1: read(x)
T2.1: write(x, 20)
T2.2: write(y, 30)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)
```~~

T1.1 -> T2.1
T2.2 -> T1.2
T2.2 -> T1.3
```

conflict graph

edge from T_i to T_j iff T_i and T_j have a conflict between them and the first step in the conflict occurs in T_i

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

```
T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3
```

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

```
T1.1 -> T2.1
T2.2 -> T1.2
T2.2 -> T1.3
```

conflict graph

edge from **T_i** to **T_j** iff **T_i** and **T_j** have a conflict between them and the first step in the conflict occurs in **T_i**

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

T2 → T1

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

T2 ⇄ T1

a schedule is conflict serializable iff it has an acyclic conflict graph

problem: how do we generate schedules that are conflict serializable? generate all possible schedules and check their conflict graphs?

solution: two-phase locking (2PL)

1. each shared variable has a lock
2. before **any** operation on a variable, the transaction must acquire the corresponding lock
3. after a transaction releases a lock, it may **not** acquire any other locks

we will usually release locks after commit or abort, which is technically *strict* two-phase locking

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)`

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)
```

solution: global ordering on locks

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)

better solution: take advantage of atomicity and abort one of the transactions!

performance improvement: allow concurrent reads with reader- and writer-locks

`acquire_reader()` and `acquire_writer()` instead of just `acquire()`

- Different types of **serializability** allow us to specify precisely what we want when we run transactions in parallel. **Conflict-serializability** is common in practice.

- **Two-phase locking** allows us to generate conflict serializable schedules. We can improve its performance by allowing concurrent reads via reader- and writer-locks.