6.033 Spring 2019
Lecture #17

• Isolation
  • Conflict serializability
  • Conflict graphs
  • Two-phase locking
goal: build reliable systems from unreliable components

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** → ?

eventually, we also want transaction-based systems to be **distributed**: to run across multiple machines
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 $T_1$, $T_2$, $\ldots$, $T_N$ concurrently, and have it “appear” as if they ran sequentially

$T_1$

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

$T_2$

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
$T_1$
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit

$T_2$
begin
write(x, 20)
write(y, 30)
commit

possible sequential schedules

$T_1$ -> $T_2$: x=20, y=30
$T_2$ -> $T_1$: 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=40

(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

T1 -\rightarrow T2: x=20, y=30
T2 -\rightarrow T1: x=20, y=40

possible sequential schedules

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)

X

at end:
x=20, y=10
(assume x, y initialized to zero)
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)
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) 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)
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)

<table>
<thead>
<tr>
<th>T2.1: write(x, 20)</th>
<th>T1.1: read(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1.1: read(x)</td>
<td>T2.1: write(x, 20)</td>
</tr>
<tr>
<td>T2.2: write(y, 30)</td>
<td>T2.2: write(y, 30)</td>
</tr>
<tr>
<td>T1.2: tmp = read(y)</td>
<td>T1.2: tmp = read(y)</td>
</tr>
<tr>
<td>T1.3: write(y, tmp+10)</td>
<td>T1.3: write(y, tmp+10)</td>
</tr>
</tbody>
</table>

T2.1 -> T1.1
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)

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)

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

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 $T_i$

$T_2$: write($x$, 20)  
$T_1$: read($x$)  
$T_2$: write($y$, 30)  
$T_1$: tmp = read($y$)  
$T_1$: write($y$, tmp+10)

$T_1$: read($x$)  
$T_2$: write($x$, 20)  
$T_2$: write($y$, 30)  
$T_1$: tmp = read($y$)  
$T_1$: write($y$, tmp+10)

$T_2$ $\rightarrow$ $T_1$  
$T_2$ $\leftrightarrow$ $T_1$

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
2PL produces a conflict-serializable schedule
(equivalently, 2PL produces a conflict graph without a cycle)

**proof:** suppose not. then a cycle exists in the conflict graph

\[ T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_k \]
2PL produces a conflict-serializable schedule
(equivalently, 2PL produces a conflict graph without a cycle)

**proof:** suppose not. then a cycle exists in the conflict graph

to cause the conflict, each pair of conflicting transactions must have some shared variable that they conflict on
2PL produces a conflict-serializable schedule
(equivalently, 2PL produces a conflict graph without a cycle)

**proof:** suppose not. then a cycle exists in the conflict graph

![Diagram showing transactions T1, T2, T3, ..., Tk and their actions on shared variables x1, x2, x3, ..., xk-1, xk.]

To cause the conflict, each pair of conflicting transactions must have some shared variable that they conflict on.

- T1 acquires x1.lock
- T2 acquires x1.lock
- T2 acquires x2.lock
- T3 acquires x2.lock
- ...
- Tk acquires xk.lock
- T1 acquires xk.lock

In the schedule, each pair of transactions needs to acquire a lock on their shared variable.
2PL produces a conflict-serializable schedule
(equivalently, 2PL produces a conflict graph without a cycle)

**proof:** suppose not. then a cycle exists in the conflict graph

To cause the conflict, each pair of conflicting **transactions** must have some **shared variable** that they conflict on.

In the schedule, each pair of **transactions** needs to acquire a lock on their **shared variable**.

In order for the schedule to progress, 

- $T_1$ must have released its lock on $x_1$
- before $T_2$ acquired it.

$T_1$ acquires $x_1.lock$
$T_1$ releases $x_1.lock$
$T_2$ acquires $x_1.lock$
$T_2$ acquires $x_2.lock$
$T_3$ acquires $x_2.lock$
...
$T_k$ acquires $x_k.lock$
$T_k$ releases $x_k.lock$
$T_1$ acquires $x_k.lock$
2PL produces a conflict-serializable schedule
(equivalently, 2PL produces a conflict graph without a cycle)

proof: suppose not. then a cycle exists in the conflict graph

\[ T_1 \xrightarrow{x_1} T_2 \xrightarrow{x_2} T_3 \xrightarrow{x_3} \ldots \xrightarrow{x_{k-1}} T_k \]

\[ x_k \]

to cause the conflict, each pair of conflicting transactions must have some shared variable that they conflict on

\[ T_1 \text{ acquires } x_1.\text{lock} \]
\[ T_1 \text{ releases } x_1.\text{lock} \]
\[ T_2 \text{ acquires } x_1.\text{lock} \]
\[ T_2 \text{ acquires } x_2.\text{lock} \]
\[ T_3 \text{ acquires } x_2.\text{lock} \]
\[ \ldots \]
\[ T_k \text{ acquires } x_k.\text{lock} \]
\[ T_1 \text{ acquires } x_k.\text{lock} \]

in the schedule, each pair of transactions needs to acquire a lock on their shared variable

in order for the schedule to progress, \( T_1 \) must have released its lock on \( x_1 \) before \( T_2 \) acquired it

\textbf{contradiction:} this is not a valid 2PL schedule
**problem:** 2PL can result in deadlock

T1
- acquire(x.lock)
- read(x)
- acquire(y.lock)
- read(y)
- release(y.lock)
- release(x.lock)

T2
- acquire(y.lock)
- read(y)
- acquire(x.lock)
- read(x)
- release(x.lock)
- release(y.lock)
<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th></th>
<th>T2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>acquire(x.lock)</td>
<td></td>
<td>acquire(y.lock)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(x)</td>
<td></td>
<td>read(y)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>acquire(y.lock)</td>
<td></td>
<td>acquire(x.lock)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(y)</td>
<td></td>
<td>read(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>release(y.lock)</td>
<td></td>
<td>release(x.lock)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>release(x.lock)</td>
<td></td>
<td>release(y.lock)</td>
<td></td>
</tr>
</tbody>
</table>

“solution”: global ordering on locks
better solution: take advantage of atomicity and abort one of the transactions!
**Performance Improvement**: Allow concurrent reads with reader- and writer-locks

### Multiple Transactions

Multiple transactions can hold reader locks for the same variable at once. A transaction can only hold a writer lock for a variable if there are *no* other locks held for that variable.
• 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.