CS162
Operating Systems and
Systems Programming
Lecture 19
Transactions, Two Phase Locking (2PL),
Two Phase Commit (2PC)

April 4, 2012
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Goals of Today’s Lecture

• Transaction scheduling

• Two phase locking (2PL) and strict 2PL

• Two-phase commit (2PC):

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

Goals of Transaction Scheduling

• Maximize system utilization, i.e., concurrency
  – Interleave operations from different transactions

• Preserve transaction semantics
  – Semantically equivalent to a serial schedule, i.e., one
    transaction runs at a time

T1: R, W, R, W

Serial schedule (T1, then T2):

Serial schedule (T2, then T1):

Two Key Questions

1) Is a given schedule equivalent to a serial execution of
transactions?


Serial schedule (T1, then T2):

Serial schedule (T2, then T1):

2) How do you come up with a schedule equivalent to a
serial schedule?

4/4
Anthony D. Joseph and Ion Stoica CS162 ©UCB Spring 2012
Lec 19.3
### 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 storage/database state, the effect (on storage/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)
  - 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...

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), R(A), W(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A),</td>
</tr>
</tbody>
</table>

- Violates transaction semantics

### Anomalies with Interleaved Execution

- Write-read conflict (reading uncommitted data)
  - 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

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A), W(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A),</td>
</tr>
</tbody>
</table>

| 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)
  
  \[
  \begin{align*}
  T1: & \, W(A), \, W(B) \\
  T2: & \, W(A), \, W(B)
  \end{align*}
  \]

- 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:
  
  \[
  \begin{align*}
  T1: & \, R(A), \, W(A), \, R(B), \, W(B) \\
  T2: & \, R(A), \, W(A), \, R(B), \, W(B)
  \end{align*}
  \]

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:
  
  \[
  \begin{align*}
  T1: & \, R(A), \, W(A), \, R(B), \, W(B) \\
  T2: & \, R(A), \, W(A), \, R(B), \, W(B)
  \end{align*}
  \]
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) |

Is this schedule serializable?

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) |

- 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) |

- 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.

Serializability ≠ Conflict Serializability

• Following schedule is not conflict serializable:

  T1: R(A),    W(A),
  T2:          W(A),
  T3:                WA

• However, the schedule is serializable since its output is equivalent with the following serial schedule:

  T1: R(A), W(A),
  T2:          W(A),
  T3:                WA

• Note: deciding whether a schedule is serializable (not conflict-serializable) is NP-complete.

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

<table>
<thead>
<tr>
<th>Lock Compatibility Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

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 dependency cycle leads to deadlock
  - Assume there is a cycle between Ti and Tj
  - Edge from Ti to Tj: Ti acquires lock first and Tj needs to wait
  - Edge from Tj to Ti: Tj acquires lock first and Ti needs to wait
  - Thus, both Ti and Tj wait for each other
  - Since with 2PL neither Ti nor Tj release locks before acquiring all locks they need → 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
- 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: share lock can be upgraded to exclusive lock

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?

<table>
<thead>
<tr>
<th>1</th>
<th>Lock_X(A)</th>
<th>&lt;granted&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A := A - 50</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Unlock(A)</td>
<td>↓ &lt;granted&gt;</td>
</tr>
<tr>
<td>6</td>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Lock_S(B)</td>
<td>&lt;granted&gt;</td>
</tr>
<tr>
<td>9</td>
<td>Lock_X(B)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>PRINT(A + B)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>B := B + 50</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>

No, and it is not serializable
Is this a 2PL Schedule?

1. Lock_X(A) <granted>
2. Read(A)
3. A := A - 50
4. Write(A)
5. Lock_X(B) <granted>
6. Unlock(A)
7. Read(A)
8. Lock_S(B)
9. Read(B)
10. B := B + 50
11. Write(B)
12. Unlock(B)
13. Unlock(A)
14. Read(B)
15. Unlock(B)
16. PRINT(A+B)

Yes, so it is serializable

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).

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

Is this a Strict 2PL schedule?

1. Lock_X(A) <granted>
2. Read(A)
3. A := A - 50
4. Write(A)
5. Lock_X(B) <granted>
6. Unlock(A)
7. Read(A)
8. Lock_S(B)
9. Read(B)
10. B := B + 50
11. Write(B)
12. Unlock(B)
13. Unlock(A)
14. Read(B)
15. Unlock(B)
16. PRINT(A+B)

No: Cascading Abort Possible
Is this a Strict 2PL schedule?

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

Deadlock

- Recall: if a schedule is not conflict-serializable, 2PL leads to deadlock, i.e.,
  - Cycles of transactions waiting for each other to release locks
- Recall: two ways to deal with deadlocks
  - Deadlock prevention
  - Deadlock detection
- 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

- Prevent circular waiting
- 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?
Deadlock Detection

- Allow deadlocks to happen but check for them and fix them if found.
- Create a wait-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph
- If cycle detected – find a transaction whose removal will break the cycle and kill it

Deadlock Detection (Continued)

- Example:

```
T1: S(A), S(D), S(B)
T2: X(B), X(C)
T3: S(D), S(C), X(A)
T4: X(B)
```

Durability and Atomicity

- How do you make sure transaction results persist in the face of failures (e.g., disk failures)?
- Replicate database
  - Commit transaction to each replica
- What happens if you have failures during a transaction commit?
  - Need to ensure atomicity: either transaction is committed on all replicas or none at all

Two Phase (2PC) Commit

- 2PC is a distributed protocol
- High-level problem statement
  - If no node fails and all nodes are ready to commit, then all nodes COMMIT
  - Otherwise ABORT at all nodes
- Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)
2PC Algorithm

- One coordinator
- N workers (replicas)

High level algorithm description
- Coordinator asks all workers if they can commit
  - If all workers reply "VOTE-COMMIT", then coordinator broadcasts "GLOBAL-COMMIT"
  - Otherwise coordinator broadcasts "GLOBAL-ABORT"
- Workers obey the GLOBAL messages

Failure Free Example Execution

Coordinator Algorithm

- Coordinator sends VOTE-REQ to all workers
- Wait for VOTE-REQ from coordinator
- If ready, send VOTE-COMMIT to all workers
- If not ready, send VOTE-ABORT to coordinator
- And immediately abort

Worker Algorithm

- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn’t receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers
- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort

State Machine of Coordinator

- Coordinator implements simple state machine

- INIT
  - Recv: START
  - Send: VOTE-REQ

- WAIT
  - Recv: VOTE-ABORT
  - Send: GLOBAL-ABORT
  - Recv: VOTE-COMMIT
  - Send: GLOBAL-COMMIT

- ABORT
- COMMIT
State Machine of workers

Dealing with Worker Failures

- How to deal with worker failures?
  - Failure only affects states in which the node is waiting for messages
  - Coordinator only waits for votes in "WAIT" state
  - In WAIT, if doesn't receive N votes, it times out and sends GLOBAL-ABORT

Dealing with Coordinator Failure

- How to deal with coordinator failures?
  - worker waits for VOTE-REQ in INIT
    - Worker can time out and abort (coordinator handles it)
  - worker waits for GLOBAL-* message in READY
    - If coordinator fails, workers must BLOCK waiting for coordinator to recover and send GLOBAL_* message
Example of Coordinator Failure #1

```
coordinator
  ^
  |
VOTE-REQ  VOTE-ABORT
  |
INIT  READY  ABORT  COMMIT
  |
worker 1
  ^
  |
timeout
  |
worker 2
  ^
  |
timeout
  |
worker 3
  ^
  |
timeout
```

Example of Coordinator Failure #2

```
coordinator
  ^
  |
VOTE-REQ  VOTE-COMMIT
  |
INIT  READY  ABORT  COMMIT
  |
worker
  ^
  |
block waiting for coordinator
  |
worker 1
  ^
  |
worker 2
  ^
  |
worker 3
  ^
  |
```

Remembering Where We Were

- All nodes use stable storage to store which state they were in.
- Upon recovery, it can restore state and resume:
  - Coordinator aborts in INIT, WAIT, or ABORT
  - Coordinator commits in COMMIT
  - Worker aborts in INIT, READY, ABORT
  - Worker commits in COMMIT

Blocking for Coordinator to Recover

- A worker waiting for global decision can ask fellow workers about their state:
  - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-`
  - Thus, worker can safely abort or commit, respectively
  - If another worker is still in INIT state then both workers can decide to abort
  - If all workers are in ready, need to BLOCK (don't know if coordinator wanted to abort or commit)
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 (2PL) and strict 2PL
  - Ensure conflict-serializability for R/W operations
  - If scheduler not conflict-serializable deadlocks
  - Deadlocks can be either detected or prevented

- Two-phase commit (2PC):
  - Ensure atomicity and durability: a transaction is committed/aborted either by all replicas or by none of them