Goals of Today’s Lecture

• Finish Transaction scheduling
• Two phase locking (2PL) and strict 2PL
• Two-phase commit (2PC)

Review: ACID Properties of Transactions

• Atomicity: all actions in the transaction happen, or none happen

• Consistency: transactions maintain data integrity, e.g.,
  – Balance cannot be negative
  – Cannot reschedule meeting on February 30

• Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency

• Durability: if a transaction commits, its effects persist despite crashes

Review: Transactions

• Group together a set of updates so that they execute atomically.

• Ensure that the database is in a consistent state before and after the transaction:
  – To move money from account A to B:
    – Debit A (read(A), write(A)), and Credit B (read(B), write(B))

• Use locks to prevent conflicts with other clients.
Review: 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


Review: Two Key Questions

1) Is a given schedule equivalent to a serial execution of transactions? (color codes the transaction T1 or T2)


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

Review: 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

Review: 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
Review: 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:

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

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

Review: 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:

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

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

Review: 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.

- Is this schedule serializable?

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

- Is it conflict serializable? Why?

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: W(A)

  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></td>
</tr>
<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"

Avoid deadlock by acquiring locks in some lexicographic order

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
Seagate 6TB(!) Drive Announced
Monday April 7th

- 7,200 RPM
- 6 platters?
- Interface transfer rate: 12Gb/s SAS 3.0

But not the first 6TB drive – Western Digital (11/4/2013)
- Filled with Helium
  (23% less drag than air)
- 7 platters

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?
  - 3000, 2950, 3050

Is this a 2PL Schedule?

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

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

1. \textbf{Lock}_X(A) \quad <\text{granted}>
2. Read(A)
3. A := A - 50
4. Write(A)
5. \textbf{Lock}_X(B) \quad <\text{granted}>
6. Unlock(A) \quad <\text{granted}>
7. Read(A)
8. \textbf{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

Is this a Strict 2PL schedule?

1. \textbf{Lock}_X(A) \quad <\text{granted}>
2. Read(A) \quad \textbf{Lock}_S(A)
3. A := A - 50
4. Write(A)
5. \textbf{Lock}_X(B) \quad <\text{granted}>
6. Unlock(A) \quad <\text{granted}>
7. Read(A)
8. \textbf{Lock}_S(B)
9. Read(B)
10. B := B + 50
11. Write(B)
12. Unlock(B) \quad <\text{granted}>
13. Unlock(A)
14. Read(B)
15. Unlock(B)
16. PRINT(A + B)

No: Cascading Abort Possible

Cascading Aborts

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

T1: \textbf{X}(A), \textbf{R}(A), \textbf{W}(A), \textbf{X}(B), \neg \textbf{X}(A)
T2: \textbf{X}(A), \textbf{R}(A), \textbf{W}(B), \neg \textbf{X}(A), \textbf{abort}

- Rollback of T1 requires rollback of T2, since T2 reads a value written by T1

- Solution: \textbf{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?

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

Quiz 19.1: Transactions

- Q1: True  False  It is possible for two read operations to conflict
- Q2: True  False  A strict 2PL schedule does not avoid cascading aborts
- Q3: True  False  2PL leads to deadlock if schedule not conflict serializable
- Q4: True  False  A conflict serializable schedule is always serializable
- Q5: True  False  The following schedule is serializable

```
T1: R(A), W(A),  R(B),  W(B)
T2:     R(A),  W(A),  R(B), W(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 (wait chain is acyclic going forward in time)
  - Wound-wait: If Ti is older, Tj aborts; otherwise Ti waits (wait chain is acyclic going backward in time)

- 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
Heartbleed

- SSL/TLS vulnerability in the OpenSSL library implementation of “heartbeats”
- RFC 6520: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension

What it does?
- Provides a keep-alive “heartbeat,” and discovers how big a packet can be sent (Path Max Transfer Unit)

Why?
- Need to periodically send data on TCP/UDP connection so NAT boxes/firewalls don’t close connection

How?

- Message format:
  ```c
  struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
  } HeartbeatMessage;
  ```
- One side sends request with random data
- Other side replies with response containing the SAME random data
- All traffic is encrypted
- Bug was an error in how the heartbeat receiver checks the message values

OpenSSL Code Analysis

- The length in the SSLv3 record is not checked!
- Later in the function:
  ```c
  buffer = OPENSSL_malloc(1 + 2 + payload + padding);
  bp = buffer;
  /* Enter response type, length and copy payload */
  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp);
  memcpy(bp, pl, payload);
  ```
- Allocate as much mem as requester asked for – up to ~64K bytes!
- Copies payload AND server memory beyond payload!
- No log entries, so no forensic information

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

Detailed Algorithm

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

Worker Algorithm
State Machine of Coordinator

- Coordinator implements simple state machine

State Machine of Workers

State Machine of Coordinator

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

Example of Worker Failure
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

Example of Coordinator Failure #2

Remembering Where We Were (Durability)

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

* - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
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)

Quiz 19.2: Distributed Execution

• Q1: True  False  Strict 2PL schedules prevent deadlock
• Q2: 2PC in a distributed system ensures (tick all that apply):
  True  False  Atomicity
  True  False  Consistency
  True  False  Isolation
  True  False  Durability
• Q3: True  False  2PC prevents workers from blocking during a commit.
• Q4: True  False  The coordinator maintains its state after a power failure.

Quiz 19.2: Distributed Execution

• Q1: True  False  Strict 2PL schedules prevent deadlock
• Q2: 2PC in a distributed system ensures (tick all that apply):
  True  False  Atomicity
  True  False  Consistency
  True  False  Isolation
  True  False  Durability
• Q3: True  False  2PC prevents workers from blocking during a commit.
• Q4: True  False  The coordinator maintains its state after a power failure.

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
  – 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