CS162
Operating Systems and Systems Programming
Lecture 14

Transactions: Two Phase Locking (2PL) and Two Phase Commit (2PC)

October 17, 2011

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

Goals for Today

- Two-phase locking
- Two-phase commit

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

Review: Transactions

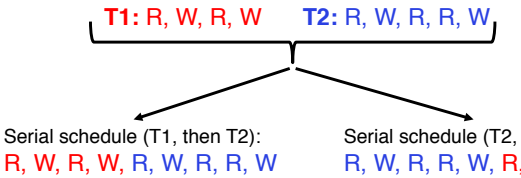
- An atomic sequence of database actions (reads/writes)
- ACID properties
  - **Atomicity:** all actions in the transaction happen, or none happens
  - **Consistency:** if each transaction is consistent, and the DB starts consistent, it ends up consistent
  - **Isolation:** execution of one transaction is isolated from that of all others
  - **Durability:** if a transaction commits, its effects persist

Review: Concurrent Transactions

- How do you run multiple concurrent transactions?
- One transaction at a time? Just execute each transaction in a critical sections?
- NO: low system utilization and large response time
  - Two transactions cannot run simultaneously even if they access different data
  - If a transaction waits for I/O operation, another transaction may not be able to use CPU

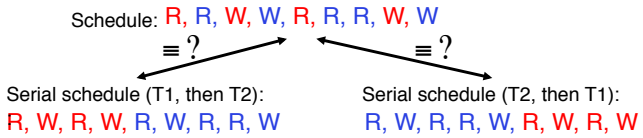
Review: Goals of Transaction Scheduling

- Maximize system utilization, i.e., concurrency
  - Interleave operations from different transactions
- Preserve transaction semantics
  - Emulate a serial schedule, i.e., one transaction runs at a time



Review: Transaction Scheduling

- Last lecture answered question:
  - Is a given schedule equivalent to a serial execution of transactions? In particular, is a given schedule **conflict-serializable**?



- This lecture: how to come up with a **conflict-seralizable** schedule?

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

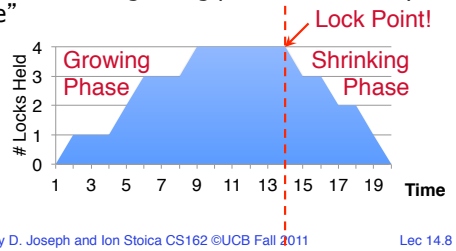
Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

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: shared 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?

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

Yes, so it is serializable

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

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

No: Cascading Abort Possible

Is this a Strict 2PL schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(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)	<granted>
11		Read(A)
12		Lock_S(B) <granted>
13		Read(B)
14		PRINT(A+B)
15		Unlock(A)
16		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

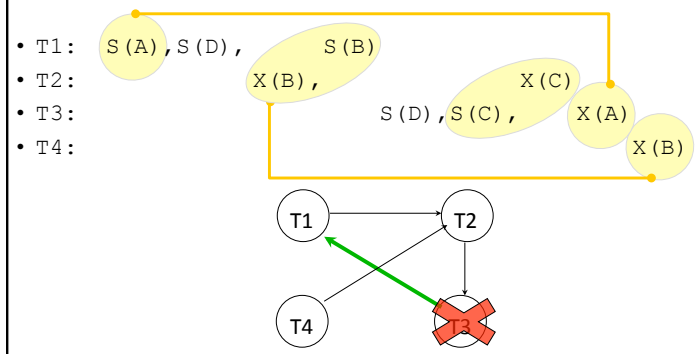
- Prevent circular waiting
- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: If Ti is older, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti is older, Tj aborts; otherwise Ti waits
- If a transaction re-starts, make sure 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

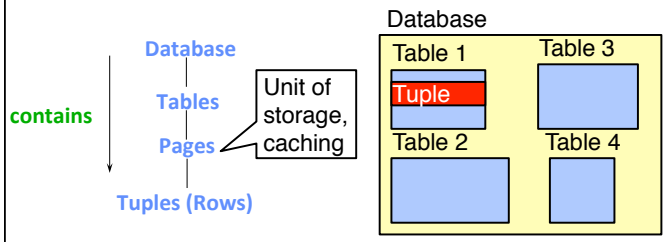
Deadlock Detection (Continued)

- Example:



Multiple-Granularity Locks

- Hard to decide what granularity to lock , e.g.,
  - Tuples (rows), pages, tables
- Shouldn't have to make same decision for all transactions!
- Data "containers" are nested:



Solution: New Lock Modes, Protocol

- Allow transactions to lock at each level, but with a special protocol using new **"intention" locks**:
- Still need S and X locks, but before locking an item, transaction **must** have proper **intention locks** on all its ancestors in the granularity hierarchy

Database
Tables
Pages
Tuples

- **IS** – Intent to get S lock(s) at finer granularity
- **IX** – Intent to get X lock(s) at finer granularity
- **SIX** mode: Like S & IX at the same time. Why useful?

	IS	IX	SIX	S	X
IS	√	√	√	√	-
IX	√	√	-	-	-
SIX	√	-	-	-	-
S	√	-	-	√	-
X	-	-	-	-	-

Multiple Granularity Lock Protocol

Database
Tables
Pages
Tuples

- Each transaction starts from the root of the hierarchy
- To get S or IS lock on a node, must hold IS or IX on parent node
  - What if transaction holds SIX on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order

Examples – 2 level hierarchy

- T1 scans table R, and updates few tuples:
  - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- T2 reads only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R
- T3 reads all of R:
  - T3 gets an S lock on R
  - OR, T3 could behave like T2; can use **lock escalation** to decide which one
  - Lock escalation dynamically asks for coarser-grain locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

The "Phantom" Problem

- With Insert and Delete, even Strict 2PL (on individual items) will not assure serializability
- Consider T1 – "Find oldest sailor"
  - T1 locks all records, and finds **oldest** sailor (*age* = 71).
  - Next, T2 inserts a new sailor; *age* = 96 and commits
  - T1 (within the same transaction) checks for the oldest sailor again and finds sailor aged 96!!
- The sailor with age 96 is a "phantom tuple" from T1's point of view – first it's not there then it is
- No serial execution where T1's result could happen!

The "Phantom" Problem – Example 2

- Consider T3 – "Find oldest sailor for each rating"
  - T3 locks all pages containing sailor records with *rating* = 1, and finds **oldest** sailor (say, *age* = 71)
  - Next, T4 inserts a new sailor; *rating* = 1, *age* = 96.
  - T4 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits
  - T3 now locks all pages containing sailor records with *rating* = 2, and finds **oldest** (say, *age* = 63)
- T3 saw only part of T4's effects!
- No serial execution where T3's result could happen!

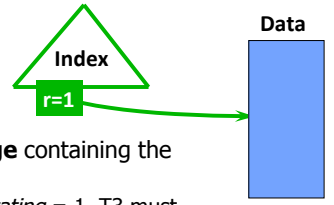
The Problem

- T1 and T3 implicitly assumed they had locked the set of all sailor records satisfying a predicate
  - Assumption only holds if no sailor records are added while they are executing!
  - Need some mechanism to enforce this assumption, e.g, **Index locking and predicate locking**
  - **Index:** data structure to allow fast access to tuples; has to be updated when a new tuple is created or removed
- Examples show that conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed!

Predicate Locking (not practical)

- Grant lock on all records that satisfy some logical predicate, e.g., *age > 2*salary*
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock
- In general, predicate locking has a lot of locking overhead

Index Locking



- T3 should lock the **index page** containing the data entries with *rating* = 1
  - If there are no records with *rating* = 1, T3 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T3 must lock table, to ensure that no records with *rating* = 1 are added or deleted

5min Break

Atomicity and Durability

- 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

Detailed Algorithm

Coordinator Algorithm

Coordinator sends **VOTE-REQ** to all workers

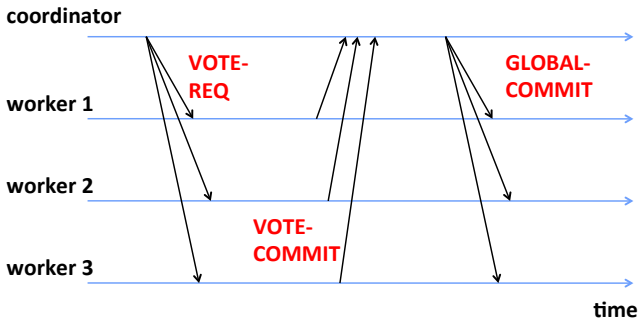
- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn't receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

- Wait for **VOTE-REQ** from coordinator
- If ready, send **VOTE-COMMIT** to coordinator
- If not ready, send VOTE-ABORT to coordinator
  - And immediately abort

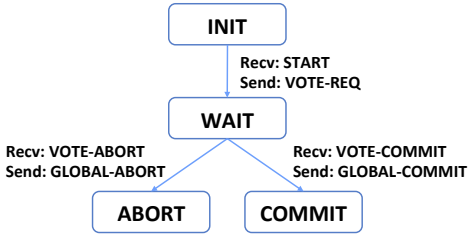
- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort

Failure Free Example Execution

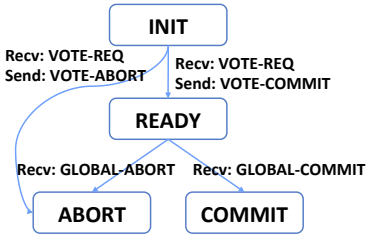


State Machine of Coordinator

- Coordinator implements simple state machine

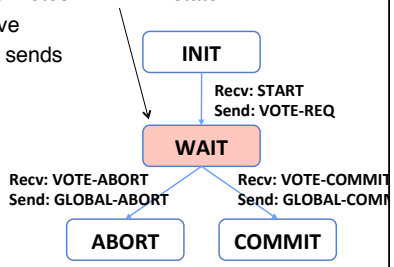


State Machine of workers

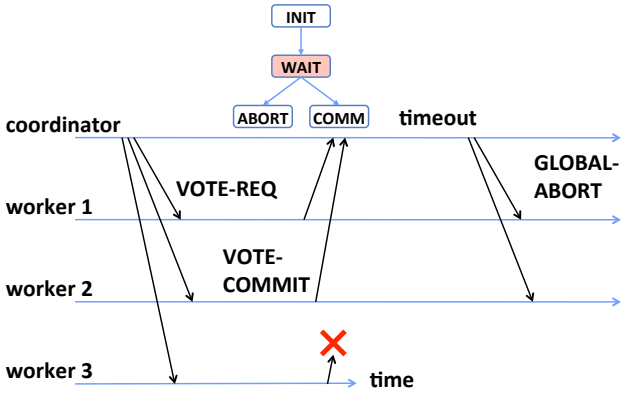


Dealing with Worker Failures

- 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

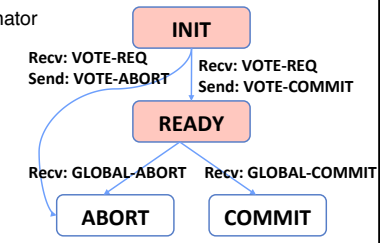


Example of Worker Failure

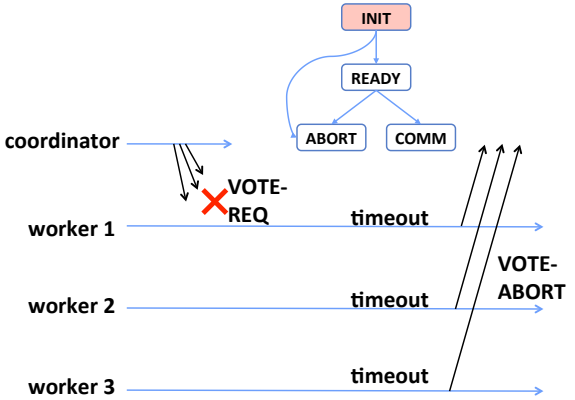


Dealing with Coordinator Failure

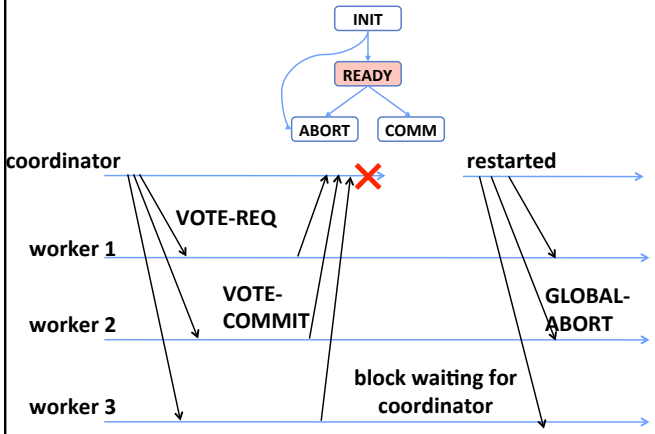
- How to deal with coordinator failures?
  - worker waits for VOTE-REQ in INIT
    - » Worker can time out and abort (coordinator handles it)
  - worker waits for GLOBAL-* message in READY
    - » If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send GLOBAL_* message



Example of Coordinator Failure #1



Example of Coordinator Failure #2

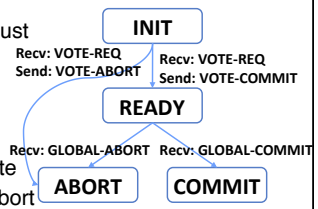


Remembering Where We Were

- 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

- Two phase locking (2PL) and strict 2PL
  - Ensure conflict-seriazability for R/W operations
  - If scheduler not conflict-serializable deadlocks
  - Deadlocks can be either prevented or prevented
- Must be careful if objects can be added or removed from databse ("phantom problem")
- Multiple granularity locking to improve concurrency
- Two-phase commit (2PC):
  - Ensure atomicity and durability: a transaction is commited/aborted either by all replicas or by none of them