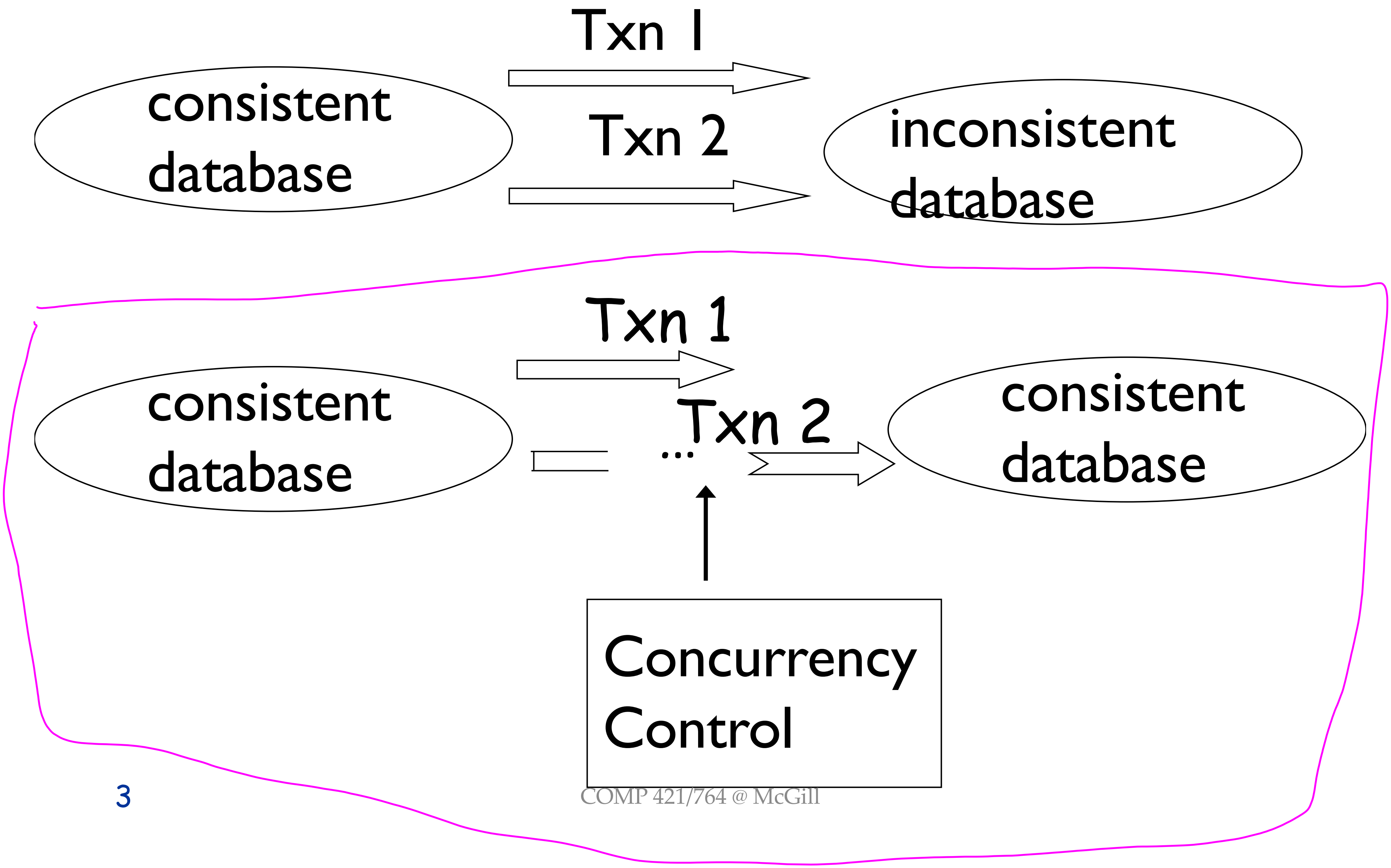
Isolation and Concurrency Control

Isolation

- DBMS can execute transactions concurrently
  - exploitation of resources
    - one transactions performs I/O, the other uses CPU etc.
    - Exploiting multi-core
- Isolation:
  - although transactions execute concurrently, each transaction runs in isolation -- not affected by the actions of other transactions
- Isolation is enforced by a **concurrency control** protocol,
- Concurrency control provides serializable executions
  - Net effect of transactions executing concurrently is identical to executing all transactions one after the other in some serial order

Isolation



Txn 1
consistent database
Txn 2
inconsistent database
Txn 1
consistent database
...
Txn 2
consistent database
Concurrency Control

Serial Execution: Example

- Consider two transactions (*Xacts/Txn*):

T1:
A=A+100,
B=B-100
commit

T2:
A=1.06*A,
B=1.06*B
commit

- T1 transfers $100 from B's account to A's account.
- T2 credits both accounts with a 6% interest payment.
- Assume A=B=200 at beginning
- Serial execution I: T1 executes before T2: Values of A and B?
  - $\boxed{A = 318, B=106}$ (Sum = 424) ←
- Serial execution II: T2 executes before T1: Values of A and B?
  - $\boxed{A = 312, B=112}$ (Sum = 424) ←
- Both execution orders make sense (sums are the same)

Concurrent Execution: Example

- Consider two transactions (*Xacts/Txn*):

T1:	T2:
A=A+100,	A=1.06*A,
B=B-100	B=1.06*B
commit	commit

Same as:

T1:	T2:
R1(A)	R2(A)
W1(A)	W2(A)
R1(B)	R2(B)
W1(B)	W2(B)
c1	c2

- T1 transfers $100 from B's account to A's account.
- T2 credits both accounts with a 6% interest payment.
- Now assume T1 and T2 are submitted at the same time
  - No guarantee that T1 will execute before T2 or vice-versa
  - The net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

- Consider two interleavings **(schedules):**

T1:	T2:
A=A+100,	
	A=1.06*A,
B=B-100	
c	
	B=1.06*B
	c

T1:	T2:
A=A+100,	
	A=1.06*A,
	B=1.06*B
	c
B=B-100	
c	

- good ✓

Same as executing one after the other

- A bad one: ✗
  - ☆ The 100$ that are transferred are included twice in the interest rate calculation

318 + 212 bad

inconsistent state

Schedules

- Transaction:
  - A sequence of read and write operations on objects of the DB (denoted as r(x)/w(x))
  - Each transaction must specify as its final action either commit (c), i.e. complete successfully or abort (a), i.e., terminate and undo all the actions carried out so far.
- Schedule:
  - sequence of actions (read,write,commit,abort) from a set of transactions
  - Reflects how the DBMS sees the execution of operations; ignores things like reading/writing from OS files etc.
- Complete Schedule:
  - Contains commit/abort for each of its transactions.
- Serial schedule:
  - Schedule where transactions are executed one after the other.

Examples

Serial Schedule

T1	T2	
R1(A)		
W1(A)		
R1(B)		
W1(B)		
c1		commit
	R2(A)	
	W2(A)	
	R2(B)	
	W2(B)	
	c2	commit

Time

Non serial Schedule

T1	T2
R1(A)	
W1(A)	
	R2(A)
	W2(A)
R1(B)	
W1(B)	
c1	
	R2(B)
	W2(B)
	c2

Time

Reading Uncommitted Data: Dirty Reads

- T1: money transfer (as before)
- T3: sum of all accounts

	T1:	T3
T1:A=A+100	R1(A) W1(A)	
T3:Read(A) T3:Read(B)		R3(A) R3(B) c3
T1:B=B-100	R1(B) W1(B) c1	

- The user perspective: appears…
  - T3 executes after T1 because it reads the A value T1 has written;
  - T3 executes before T1 because it read the B value before T1 has written

- If T3 reads from T1 before T1 commits, it might read inconsistent data (**Inconsistent or Dirty Reads**)

Reading Uncommitted Data: Dirty Reads

- T1: money transfer (as before)
- T3: sum of all accounts

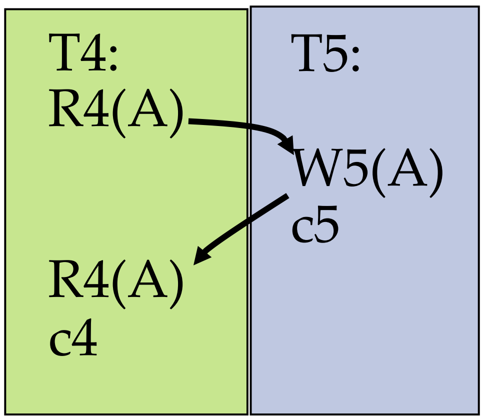
	T1:	T3:
A=A+100	R1(A)	
	W1(A)	
Read(A)		R3(A)
B=B-100	R1(B)	
	W1(B)	
	c1	
Read(B)		R3(B)
		c3

- The user perspective:
  - ☆ T3 always reads after T1 writes;
  - ☆ It is as if T3 executes serially after T1

- If T3 reads from T1 before T1 commits, it **might** read inconsistent data (**Inconsistent or Dirty Reads**)
- But it might also be ok

Unrepeatable Reads

- T4 reads A twice
- T5 updates A



- The user perspective:
  - ☆ T4 executes before T5 because it reads A before T5 writes it
  - ☆ T4 executes after T5 because it reads A after T5 writes it

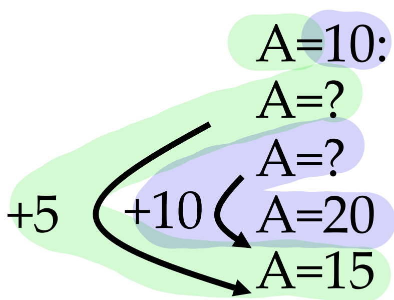
conceptually executes before AND after

- If T4 reads twice the same data item, but T5 changes the value between the first and second read, then we have **unrepeatable read** situation.

Serious

Lost Update

- T6: A=A+5
- T7: A=A+10
- A=10 at start



T6:	T7:
R6(A) (10)	
	R7(A) (10)
	W7(A) (+10)
	c7 A=20
W6(A) (+5)	
c6 A=15	

- The user perspective:
  - It is as if T7's update has never taken place; it is not reflected in the database
  - If it were reflected the final value of A should be 25 and not 15

❑ Can lead to **lost update**

T7's update was lost

Committed and Aborted Transactions

- If a transaction aborts, all its actions are undone.
- It is if they were never carried out

T6:	T8:
R6(A)	
W6(A)	
	R8(A)
	W8(B)
	W8(A)
	c8
$W^{-1}6(A)$	
abort	

- The user perspective:
  - T8 reads a value for A that actually will never exist!
  - Problem even bigger if the read triggered a further change in the database
  - Problem even bigger if A is updated in between; how to undo in this case???

- **Dirty Read** can lead to reading a non-existing value
- **Dirty Write** can mess up the database
  - AVOID AT ALL COSTS!!

rewriting before a commit

bad

Conflicting Operations

- Conflicting operations: Two operations conflict if (from 2 different transactions)
  - They access the same object
  - Both operations are writes, or one is write and one is read
- Schedules
  - serial schedule: T1 T2 = r1(x) r1(y) w1(y) c1 r2(x) w2(x) r2(y) c2
    - all operations of T1 are executed before any operation of T2
    - in particular: if T1 and T2 have several conflicting operations, T1's operation is always executed before T2's conflicting operation.
  - Our examples:
    - one operation of T1 was ordered before the conflicting operation of T2
    - another operation of T1 was ordered after T2's conflicting operation

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same (committed) transactions
  - Every pair of conflicting actions of (committed transactions) is ordered the same way
- Schedule S is conflict serializable if
  - S is conflict equivalent to some serial schedule which contains the committed transactions of S
  - Textbook differentiates between
    - Serializable
    - Conflict-serializable
    - View-serializable
  - Here:
    - conflict-serializable = serializable
    - Ignore view-serializable

Serial

Examples

SL1: T1	T2
r1(x)	
w1(x)	
w1(y)	
c1	
	r2(z)
	r2(y)
	w2(x)
	c2

conflicts ordered in same way

SL2: T1	T2
	r2(z)
	r2(y)
	w2(x)
	c2
r1(x)	
w1(x)	
w1(y)	
c1	

Serial Schedules

Schedules written in a different format:

SL1: r1(x) w1(x) w1(y) c1 r2(z) r2(y) w2(x) c2

SL2: r2(z) r2(y) w2(x) c2 r1(x) w1(x) w1(y) c1 interleaved

Examples

conflict serializable

S1: T1	T2
r1(x)	
w1(x)	
	r2(z)
	r2(y)
	w2(x)
	c2
w1(y)	
c1	

S2: T1	T2
r1(x)	
	r2(z)
	r2(y)
w1(x)	
w1(y)	
c1	
	w2(x)
	c2

S3: T1	T2
r1(x)	
	r2(z)
w1(x)	
w1(y)	
	r2(y)
c1	
	w2(x)
	c2

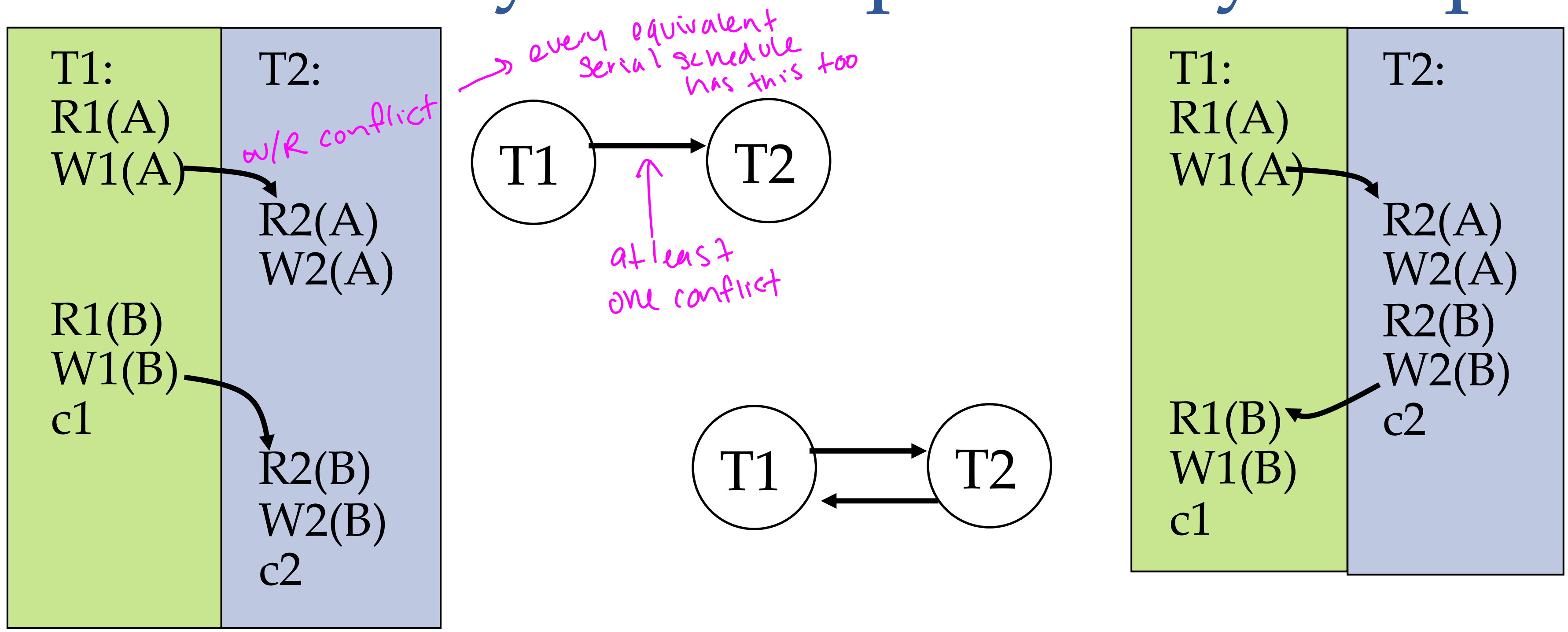
Schedules written in a different format:

S1: r1(x) w1(x) r2(z) r2(y) w2(x) c2 w1(y) c1

S2: r1(x) r2(z) r2(y) w1(x) w1(y) c1 w2(x) c2

S3: r1(x) r2(z) w1(x) w1(y) c1 r2(y) w2(x) c2

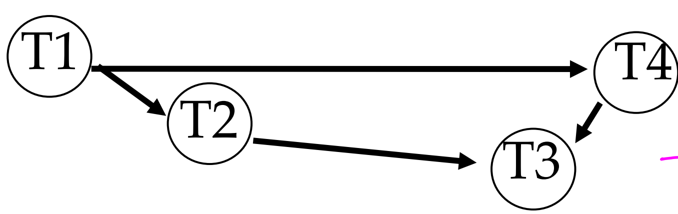
Serializability and Dependency Graphs



- Dependency graph / Serialization graph / precedence graph / Serializability graph for a schedule:
  - Let S be a schedule (T, O, <)
    - Each transaction Ti in T is represented by a node
    - There is an edge from Ti to Tj if an operations of Ti precedes and conflicts with one of Tj's operations in the schedule.

Dependency Graphs

- <u>Theorem</u>: Schedule is conflict serializable if and only if its dependency graph is acyclic
- Generating an equivalent serial schedule (flatten)
  - Continue until no nodes are left
    - Choose a source (i.e. a node without incoming edges)
      - put the corresponding transaction next in the serial order
      - Delete the node and all outgoing edges



flatten →

T1 -> T2 -> T4 -> T3
T1 -> T4 -> T2 -> T3

We want an equivalent serial schedule

Concurrency Control

- Given an execution (schedule) we can test whether the execution was serializable
  - If execution was serializable, then ok
  - If not serializable, then it's too late!
- Concurrency control:
  - during execution take measures such that a non-serializable execution can never happen

prevent

graphs are just to understand

we need to prevent non-serializable executions

Concurrency Control: Locking

- No conflict: transactions can execute at the same time (e.g. all reads)
- Upon first conflict: the second transaction has to wait until the first transaction commits/aborts
- Locks: Two types, because two read operations do not conflict
- Basics of locking:
  - Each transaction Ti must obtain a S (*shared*) lock on object *before* reading, and an X (*exclusive*) lock on object *before* writing.
  - If an X lock is granted on object O, no other lock (X or S) might be granted on O at the same time. If one is writing, everyone must wait
  - If an S lock is granted on object O, no X lock might be granted on O at the same time. If one is reading, writes must wait
  - If a conflicting lock is active, the transaction must wait until the lock is released
  - Conflicting locks are expressed by the compatibility matrix:

at end of transaction

	S	X
S	✓	--
X	--	--

Strict Two Phase Locking

- Phase I = growing phase: acquiring locks whenever you need one
  - Each transaction Ti must request a S (*shared*) lock on object *before* reading, and an X (*exclusive*) lock on object *before* writing.
  - If no conflicting lock is active is set, the lock can be granted (and the transaction can execute the operation),
  - If a conflicting lock is active the transaction must wait until the lock is released
- Phase 2 = shrinking phase: After a transaction has released one of its lock (unlock) it may not request any further locks
- Strict: a transactions releases all its lock at the end of its execution after commit
  - Shrinking phase happens in one shot at end of transaction.

T1:	T2:
S1(A)	
R1(A)	
X1(B)	
W1(B)	
	S2(A)
	R2(A)
	S2(B)
	Blocked
c1	
U1(A)	
~~S1(D)~~	
U1(B)	
	R2(B) (unblocked)
	C2
	U2(A)
	U2(B)

2PL allows only serializable schedules
Strictness: No dirty reads / no dirty writes

Strict 2PL: another example

T1: R(A), W(B) | T2: W(A) | T3: R(A), R(B)

Order of submission: R1(A), R3(A), W1(B), W2(A), R3(B)

T1:	T2:	T3:
S1(A)		
R1(A)		S3(A)
X1(B)		R3(A)
W1(B)		
	X2(A)	
		S3(B)
c1		
U1(A,B)		
		R3(B)
		C3
		U3(A,B)
	W2(A)	
	C2	
	U2(A)	

Lock Table: empty at start

- A: T1:S
- A: T1:S T3:S
- A: T1:S T3:S — B: T1:X
- A: T1:S T3:S; WL T2:X — B: T1:X
- A: T1:S T3:S; WL T2:X — B: T1:X; WL T3:S
- A: T3:S; WL T2:X — B: T3:S
- A: T2:X

Lock Table: empty at end

Implementing strict 2PL

- Lock request
  - If lock is S, no X lock is active and the request queue is empty:
    - add the lock to the granted lock queue and set lock type to S
  - If lock is X and no lock active (=> the request queue is also empty):
    - add the lock to the granted lock queue and set lock type to X
  - Otherwise:
    - add the lock to the request lock queue
  - In the first two cases, the transaction can continue immediately. In the last case the transaction is blocked until the lock is granted
- Lock release (at end of transaction)
  - Remove the lock from the granted lock queue
  - If this was the only lock granted on the object:
    - grant one write lock (if the first lock in the request queue is a write) or
    - n read locks (if the first n locks in the request queue are reads) as described above.

Details

- A transaction does not request the same lock twice.
- A transaction does not need to request a S lock on an object for which it already holds an X lock.
- If a transaction has an S lock and needs an X lock it must wait until all other S locks (except its own) are released

Implementation Details

- Locks are managed using a **lock table**
- The lock table has a lock table entry for each object that is currently locked
  - Pointer to queue of granted locks (or simply the number of transactions currently holding a lock)
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests (waiting transactions)
  - A transaction T contains only one lock per object
    - if a T has an S lock and requests an X lock, the S lock is upgraded to an X lock (may have to wait/block for other S locks to be released)
- Locking and unlocking have to be atomic operations
  - Set latch/semaphore when accessing lock table
- Transaction table:
  - For each transaction T contains pointer to a list of locks held by T

Why does 2PL work?

- When is a schedule not serializable?
  - If there are operations of transactions T1 and T2 such that T1 should be ordered before T2 AND after T2 in the schedule T1 -> T2 -> T1

R1/W2 + R2/W1
R1/W2 + W2/W1
R1/W2 + W2/R1
W1/W2 + R2/W1
W1/W2 + W2/W1
…
W1/R2 + R2/W1
…

Submission order 1
R1(A) W2(A) W2(B) W1(B)

T1:	T2:
S1(A)	
R1(A)	
	X2(A)
	Blocked
X1(B)	
W1(B)	
C1	
U(A,B)	
	W2(A)
	X2(B)
	…

transactions ordered according to who got the first lock

Submission order 2
R1(A) W2(B) W1(B) W2(A)

T1:	T2:
S1(A)	
R1(A)	
	X2(B)
	W2(B)
X1(B)	
Blocked	
	X2(A)
	blocked

Deadlock: see later

Why does simple locking NOT work?

- It allows any order!!

Submission order 1

R1(A) W2(A) W2(B) W1(B)

R1/W2 + R2/W1
R1/W2 + W2/W1
R1/W2 + W2/R1
W1/W2 + R2/W1
W1/W2 + W2/W1
…
W1/R2 + R2/W1
…

T1:	T2:
S1(A)	
R1(A)	
U1(A)	
	X2(A)
	W2(A)
	U2(A)
	X2(B)
	W2(B)
	U2(B)
	C2
X1(B)	
…	

Non serializable!

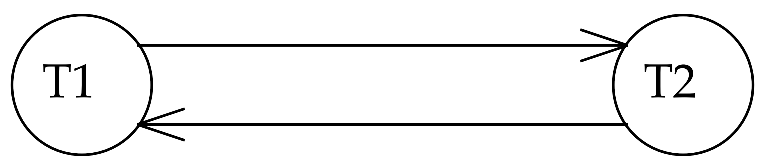
Same as setting no locks at all

Operations are ordered as they are submitted

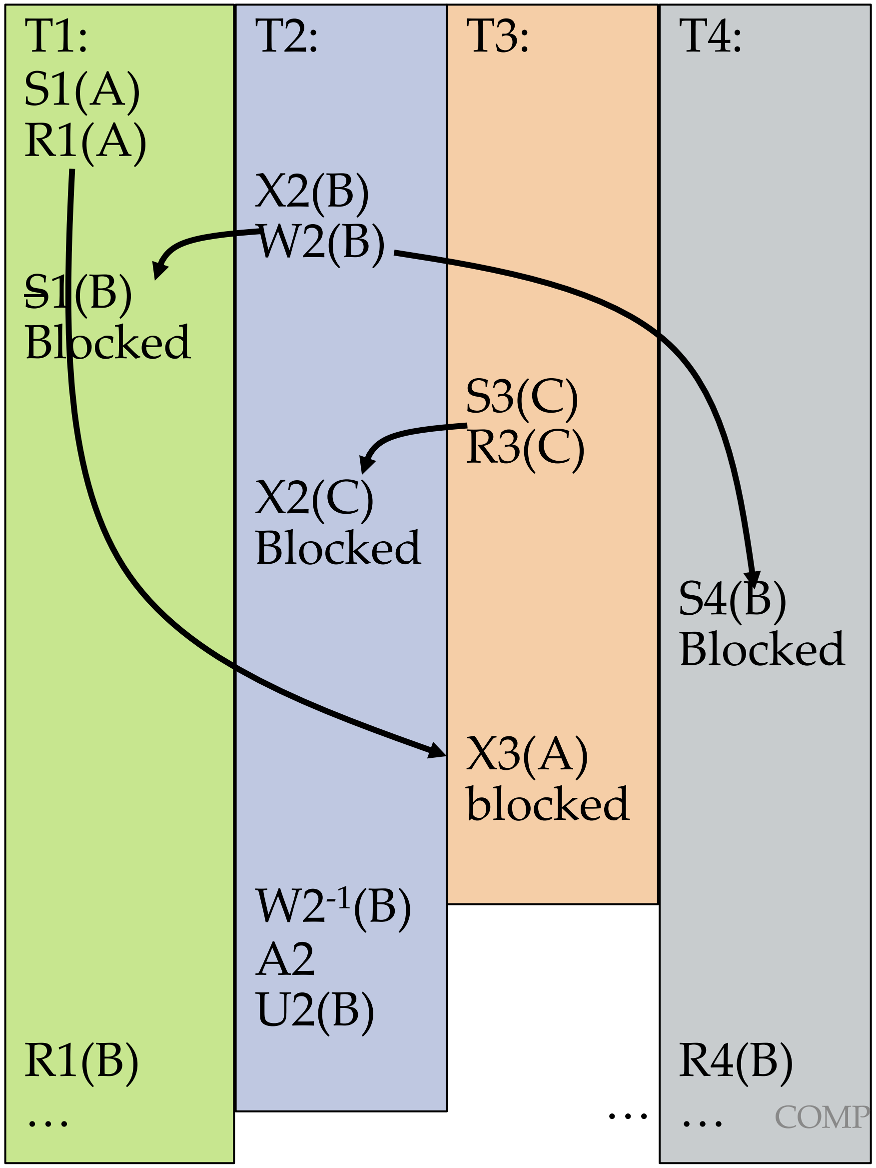
Deadlocks

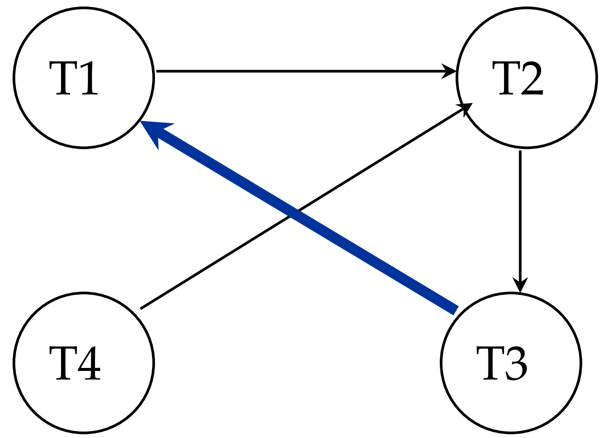
- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Deadlock detection: look for cycles in the wait-for graph

T1:	T2:
S1(A)	
R1(A)	
	X2(B)
	W2(B)
X1(B)	
Blocked	
	X2(A)
	blocked



Deadlock Detection (Continued)



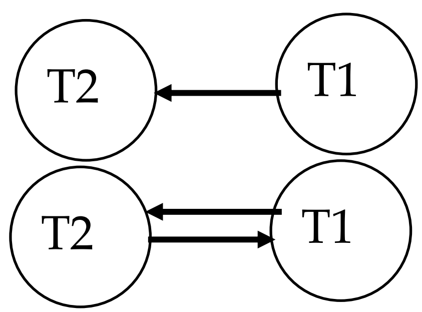


Dependency graph - wait-for-graph

- Note: is similar to dependency graph with the following difference
  - If there is an edge from T2 to T1 in the wait-for-graph, then T2's operation will execute after T1's operation (T2 waits for T1 to release the lock), hence, in the dependency graph there is an edge from T1 to T2
  - Deadlocks can happen because 2PL avoids unserializable schedules by locking objects!

T1:	T2:
S1(A)	
R1(A)	
	X2(B)
	W2(B)
X1(B)	
Blocked	
	X2(A)
	blocked

Wait-for-graph



Depend. graph