CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #16: Transactions 2: 2PL and Deadlocks

Reminders

- On Thursday Nov 1
  - PA2 due
  - PA3 and HW7 out
  - Recitation for PA3 by Deepika – don’t miss!
- On Tuesday Nov 6
  - No lecture, but Deepika will be in the classroom during lecture time for extra office hours.

Review (last lecture)

- DBMSs support ACID Transaction semantics.
- Concurrency control and Crash Recovery are key components

Review

- For Isolation property, serial execution of transactions is safe but slow
  – Try to find schedules equivalent to serial execution
- One solution for "conflict serializable" schedules is Two Phase Locking (2PL)

Outline

- **2PL/2PLC**
- Lock Management
- Deadlocks
  – detection
  – Prevention
- Specialized Locking

Serializability in Practice

- DBMS does not test for conflict serializability of a given schedule
  - Impractical as interleaving of operations from concurrent Xacts could be dictated by the OS
- Approach:
  - Use specific protocols that are known to produce conflict serializable schedules
  - But may reduce concurrency

Solution?

- One solution for “conflict serializable” schedules is Two Phase Locking (2PL)

Answer

- (Full answer:) use locks; keep them until commit ('strict 2 phase locking')
- Let's see the details

Lost update problem - no locks

T1	T2
Read(N)	
	Read(N)
N = N -1	
	N = N -1
Write(N)	
	Write(N)

Solution – part 1

- with locks:
- lock manager: grants/denies lock requests

Lost update problem – with locks

time	T1	T2	lock manager
	lock(N)		grants lock
		lock(N)	denies lock
	Read(N)	T2: waits	
	N=N-1		
	Write(N)		
	Unlock(N)		grants lock to T2
		Read(N) ...	

Locks

- Q: I just need to read 'N' - should I still get a lock?

VirginiaTech

Solution – part 1

- Locks and their flavors
  - exclusive (or write-) locks
  - shared (or read-) locks
  - <and more ... >
- compatibility matrix

T2 wants / T1 has	S	X
S		
X		

Solution – part 1

- Locks and their flavors
  - exclusive (or write-) locks
  - shared (or read-) locks
  - <and more ... >
- compatibility matrix

T2 wants / T1 has	S	X
S	Yes	
X		

Solution – part 1

- transactions request locks (or upgrades)
- lock manager grants or blocks requests
- transactions release locks
- lock manager updates lock-table

Solution – part 2

locks are not enough – eg., the ‘inconsistent analysis’ problem

‘Inconsistent analysis’

time

T1	T2
Read(A)	
A=A-10	
Write(A)	
	Read(A)
	Sum = A
	Read(B)
	Sum += B
Read(B)	
B=B+10	
Write(B)	

'Inconsistent analysis' – w/ locks

time	T1	T2
	L(A)	
	Read(A)	
	...	
	U(A)	
		L(A)
		....
		L(B)
		....

the problem remains!

T2 reads an inconsistent DB state

Solution??

General solution:

- Protocol(s)
- Most popular protocol: 2 Phase Locking (2PL)

2PL

X-lock version: transactions issue no lock requests, after the first 'unlock'

THEOREM: if **ALL** transactions in the system obey 2PL --> all schedules are serializable

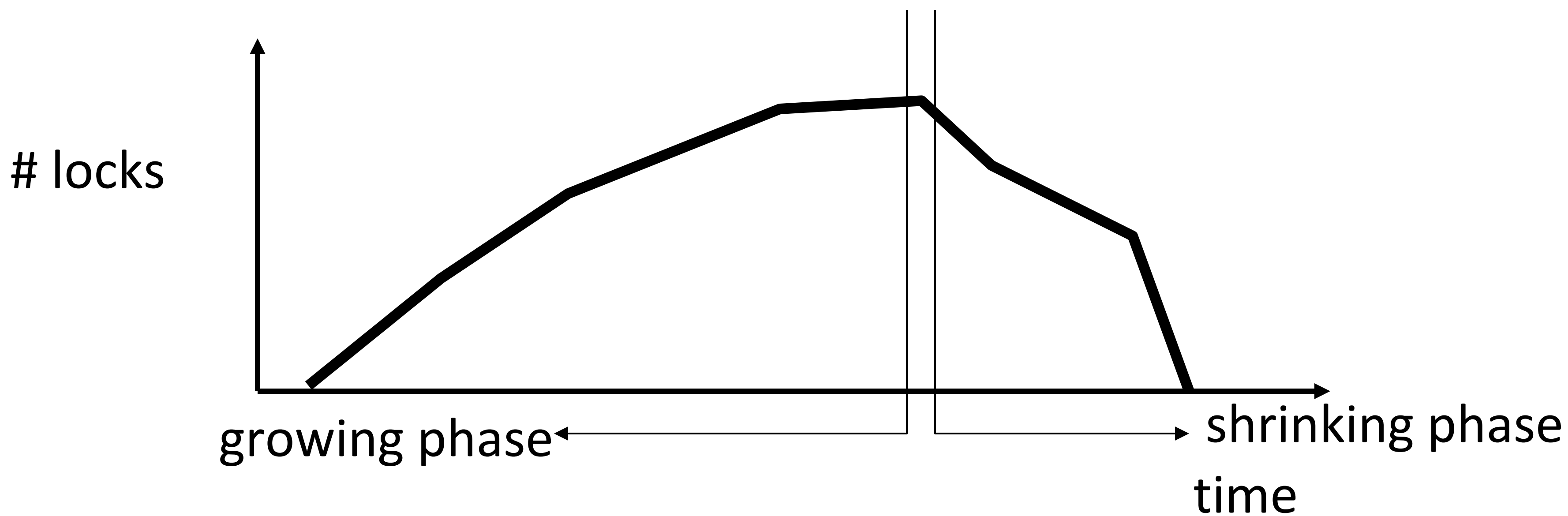
2PL – example

- ‘inconsistent analysis’ – how does 2PL help?
- how would it be under 2PL?

2PL – X/S lock version

transactions issue no lock/upgrade request, after the first unlock/downgrade

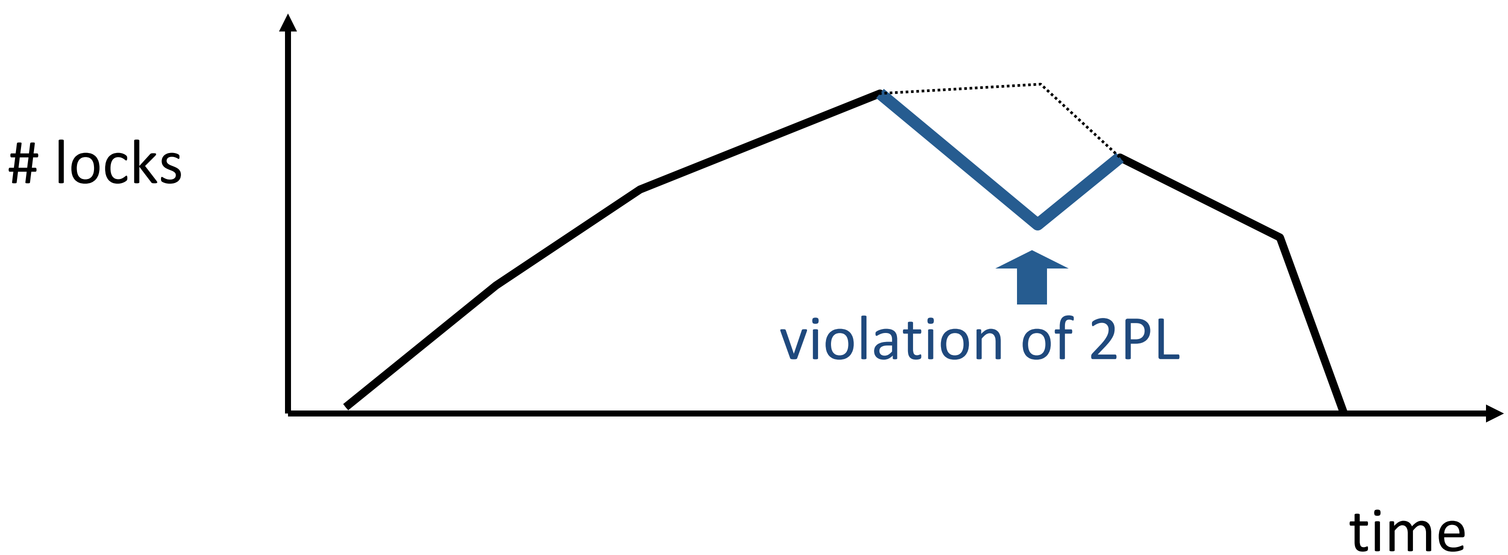
In general: ‘growing’ and ‘shrinking’ phase



2PL – X/S lock version

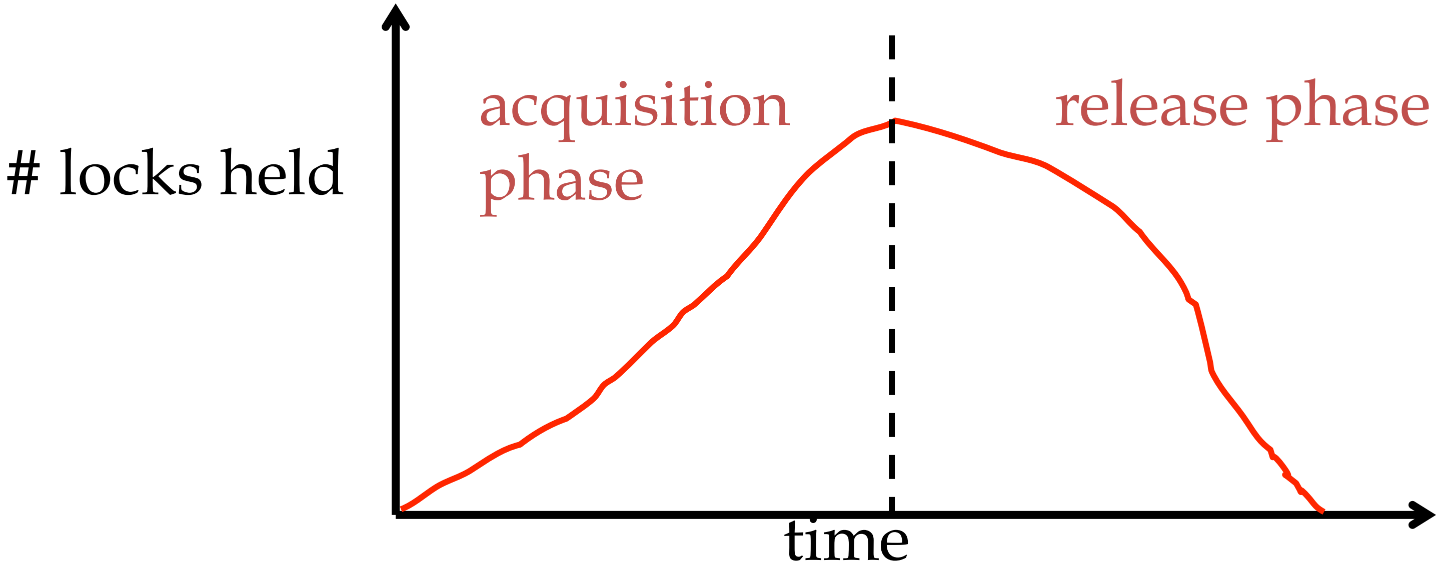
transactions issue no lock/upgrade request, after the first unlock/downgrade

In general: 'growing' and 'shrinking' phase



Two-Phase Locking (2PL), cont.

- 2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic), but, it is subject to Cascading Aborts.



2PL

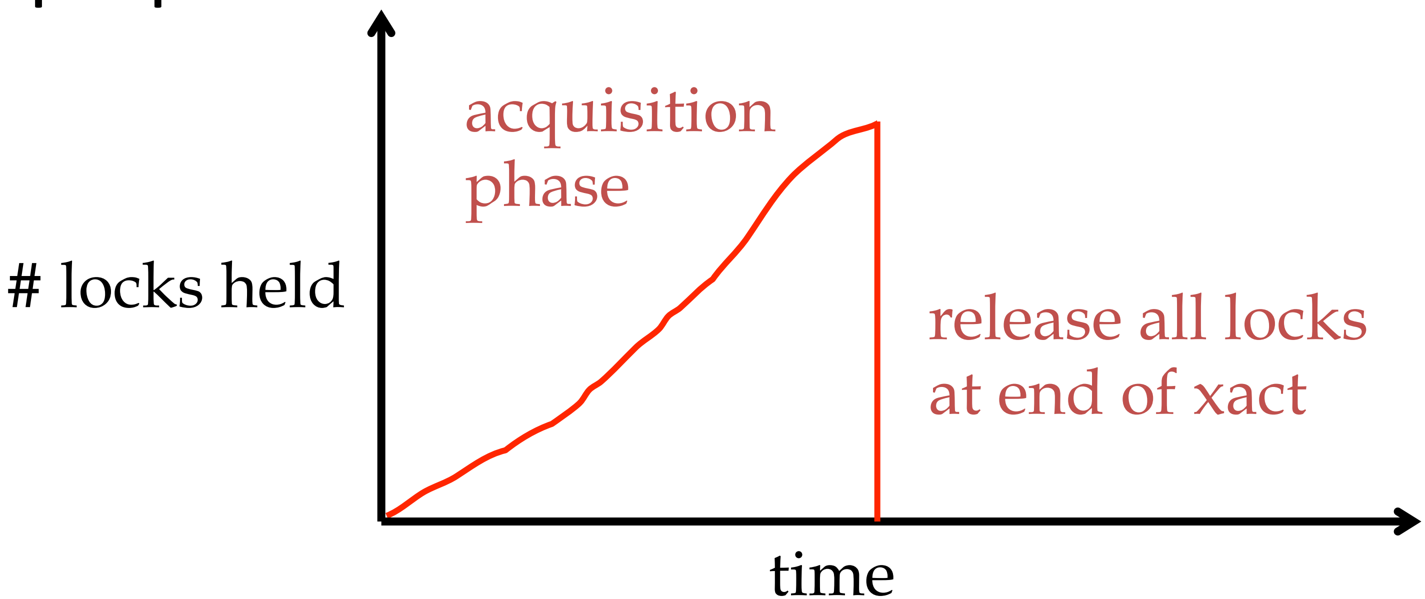
- Problem: Cascading Aborts
- Example: rollback of T1 requires rollback of T2!

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A)	

- Solution: Strict 2PL, i.e,
- keep all locks, until 'commit'

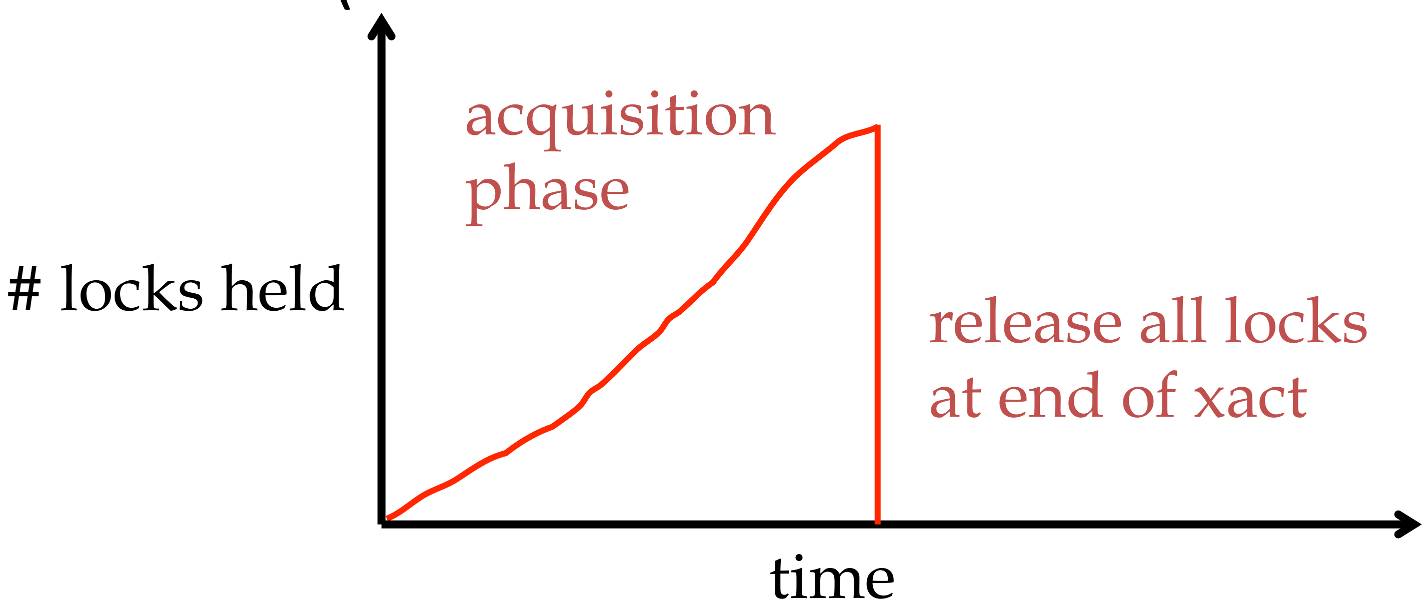
Strict 2PL

- Allows only conflict serializable schedules, but it is actually stronger than needed for that purpose.



Strict 2PL == 2PLC (2PL till Commit)

- In effect, “shrinking phase” is delayed until
  - Transaction commits (commit log record on disk), or
  - Aborts (then locks can be released after rollback).



Non-2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	
A: = A-50	
Write(A)	
Unlock(A)	
	Lock_S(A)
	Read(A)
	Unlock(A)
	Lock_S(B)
	Read(B)
	Unlock(B)
	PRINT(A+B)
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

2PL, A= 1000, B=2000, Output =?

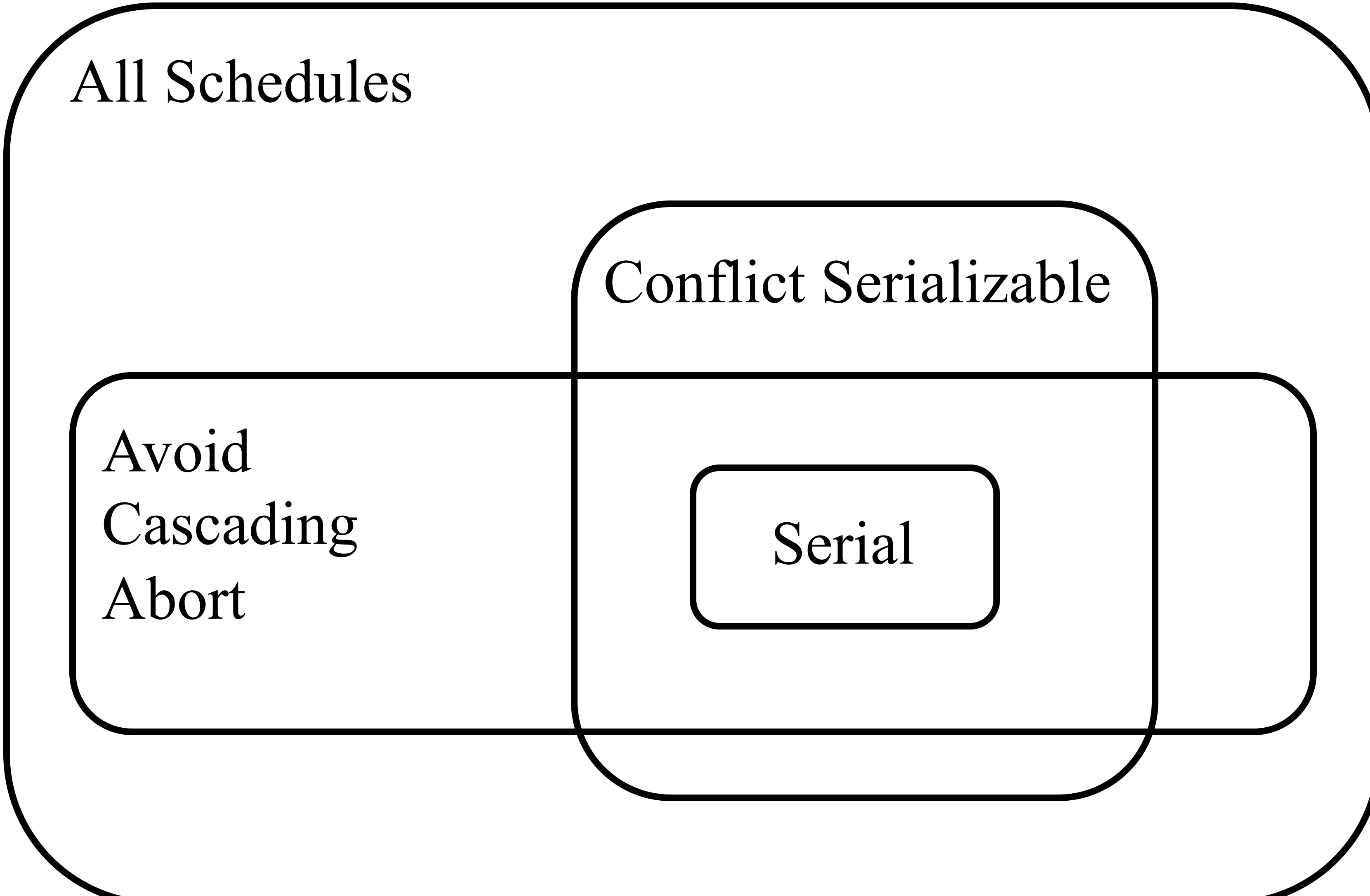
Lock_X(A)	
Read(A)	
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Lock_S(A)
	Read(A)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	
	Lock_S(B)
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

VirginiaTech

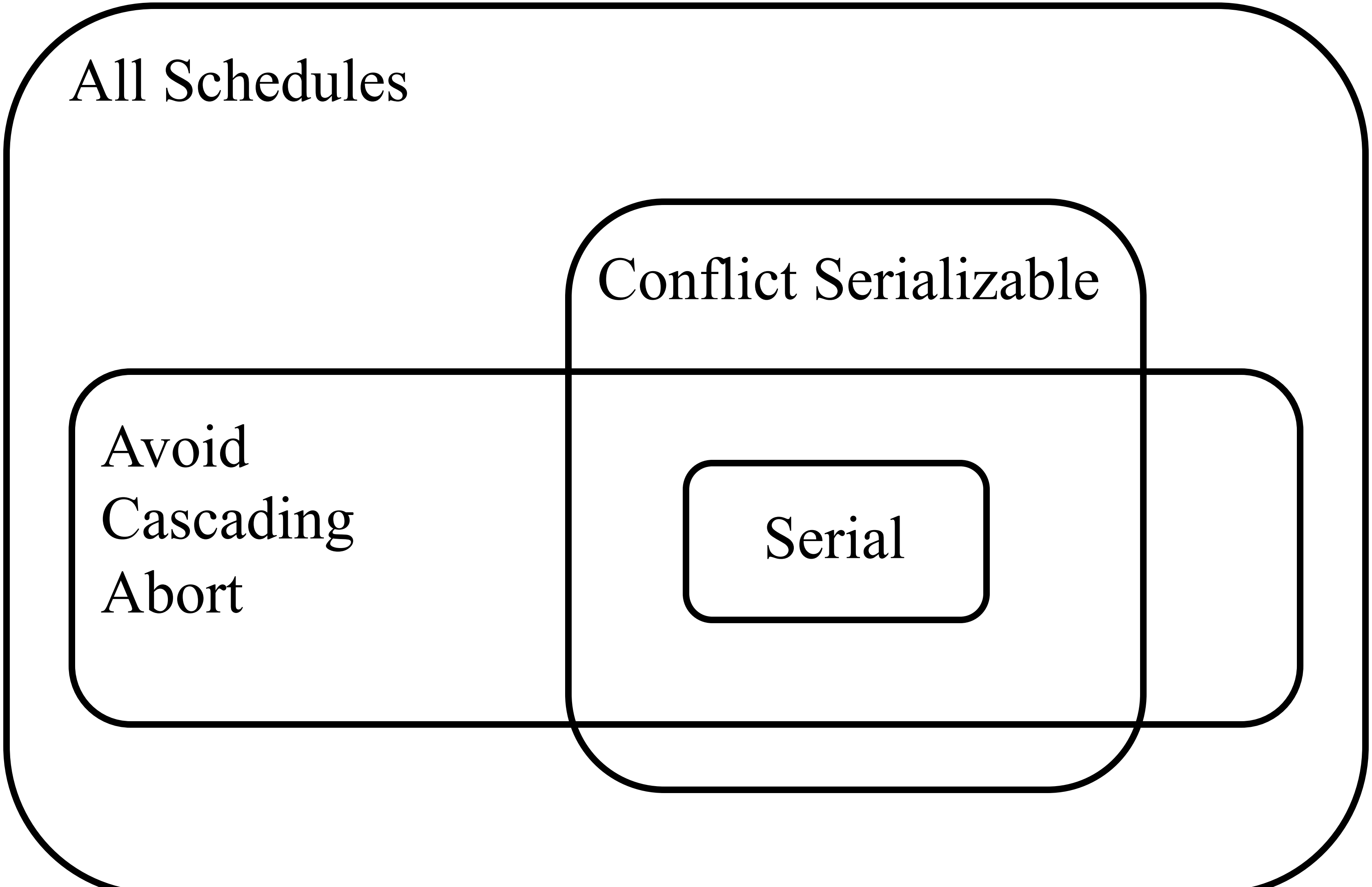
Strict 2PL, A= 1000, B=2000, Output =?

Lock_X(A)	
Read(A)	
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Lock_S(A)
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

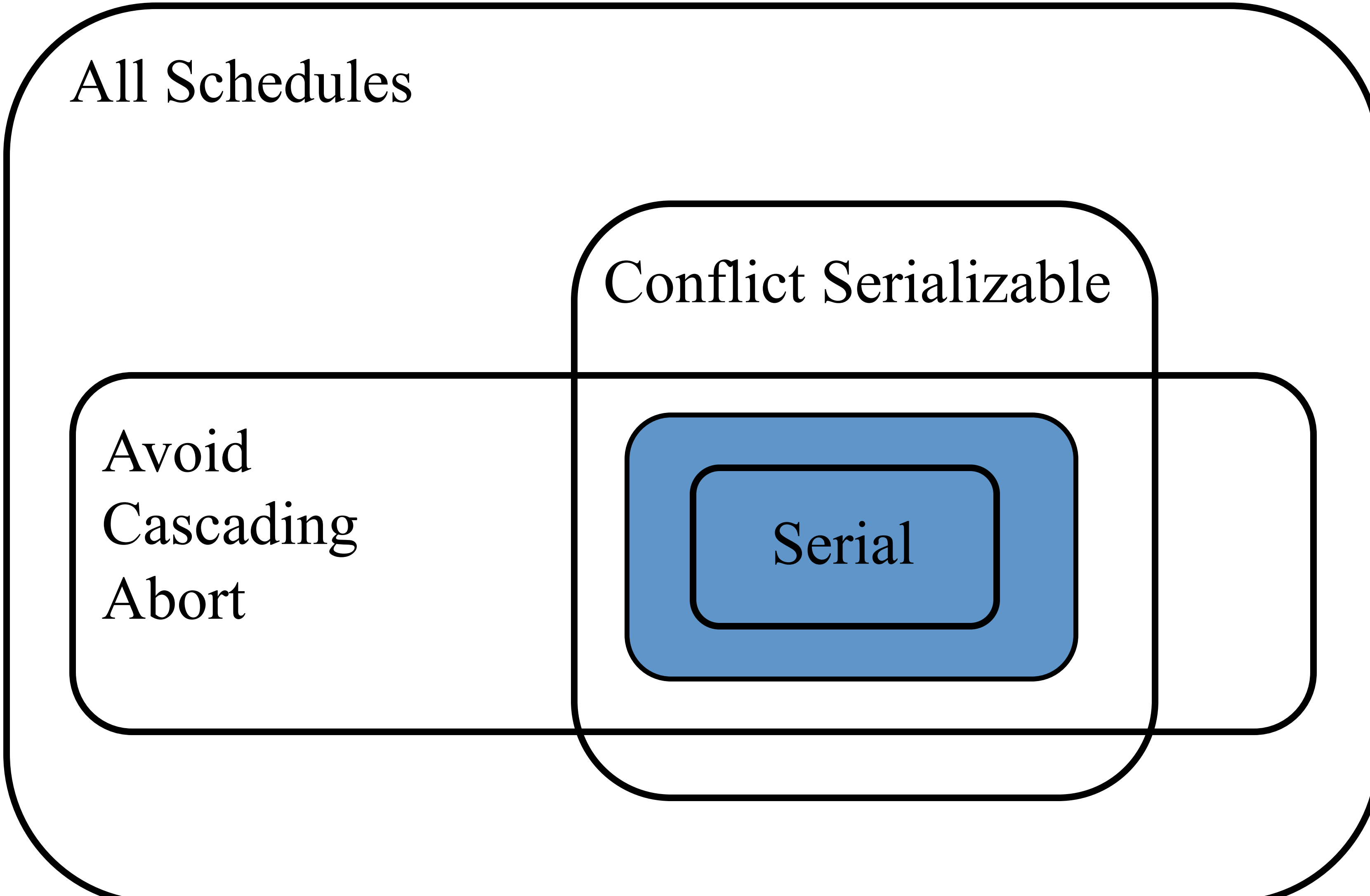
Venn Diagram for Schedules



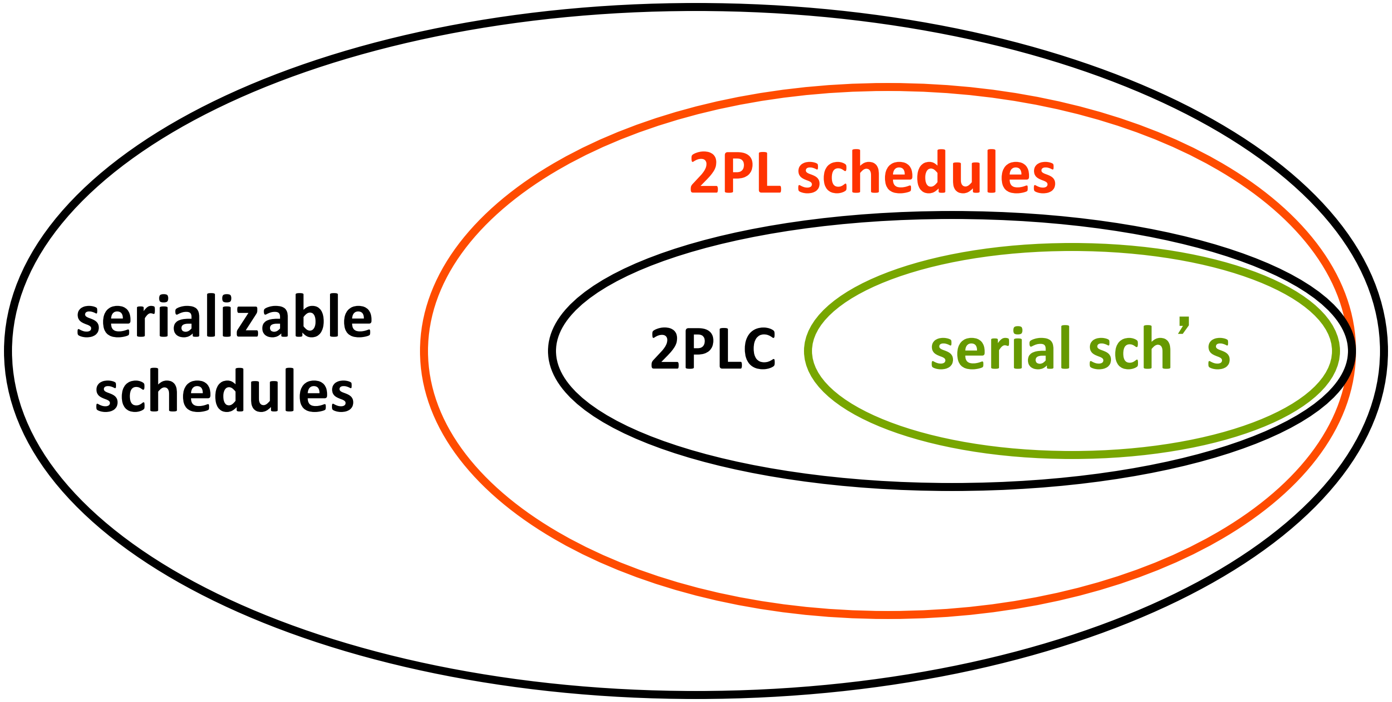
Q: Which schedules does Strict 2PL allow?



Q: Which schedules does Strict 2PL allow?



Another Venn diagram



Outline

- 2PL/2PLC
- **Lock Management**
- Deadlocks
  - detection
  - Prevention
- Specialized Locking

Lock Management

- Lock and unlock requests handled by the Lock Manager (LM).
- LM contains an entry for each currently held lock.
- Q: structure of a lock table entry?

Lock Management

- Lock and unlock requests handled by the Lock Manager (LM).
- LM contains an entry for each currently held lock.
- Lock table entry:
  - Ptr. to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests

Lock Management, cont.

- When lock request arrives see if any other xact holds a conflicting lock.
  – If not, create an entry and grant the lock
  – Else, put the requestor on the wait queue
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Lock Management, cont.

- Two-phase locking is simple enough, right?
- We’re not done. There’s an important wrinkle …

Example: Output = ?

Lock_X(A)	
	Lock_S(B)
	Read(B)
	Lock_S(A)
Read(A)	
A: = A-50	
Write(A)	
Lock_X(B)	

Example: Output = ?

lock mgr:

Lock_X(A)		grant
	Lock_S(B)	grant
	Read(B)	
	Lock_S(A)	wait
Read(A)		
A: = A-50		
Write(A)		
Lock_X(B)		wait

Outline

- Lock Management
- **Deadlocks**
  - **detection**
  - Prevention
- Specialized Locking

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  – Deadlock prevention
  – Deadlock detection
- Many systems just punt and use Timeouts
  – What are the dangers with this approach?

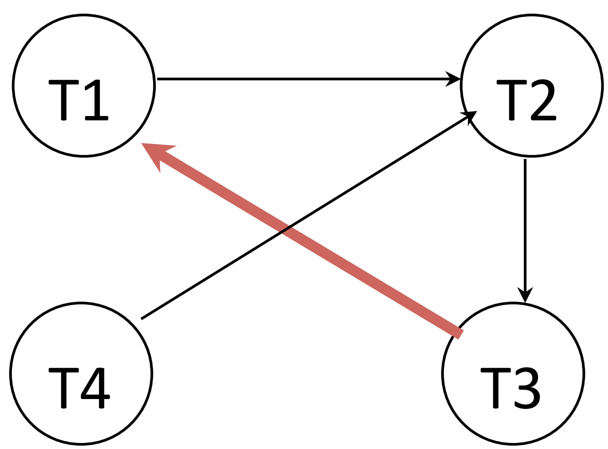
Deadlock Detection

- Create a waits-for graph:
  – Nodes are transactions
  – Edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in waits-for graph

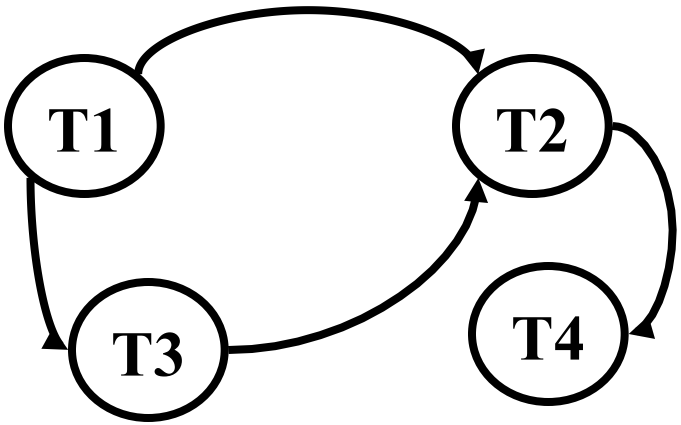
Deadlock Detection (Continued)

Example:

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

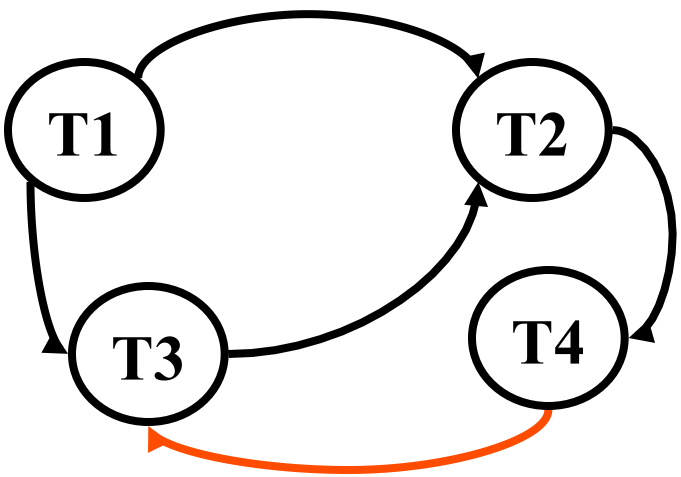


Another example



- is there a deadlock?
- if yes, which xacts are involved?

Another example

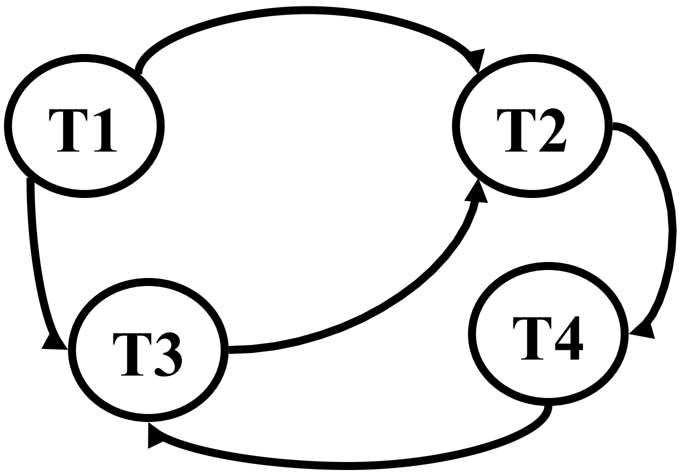


- now, is there a deadlock?
- if yes, which xacts are involved?

Deadlock detection

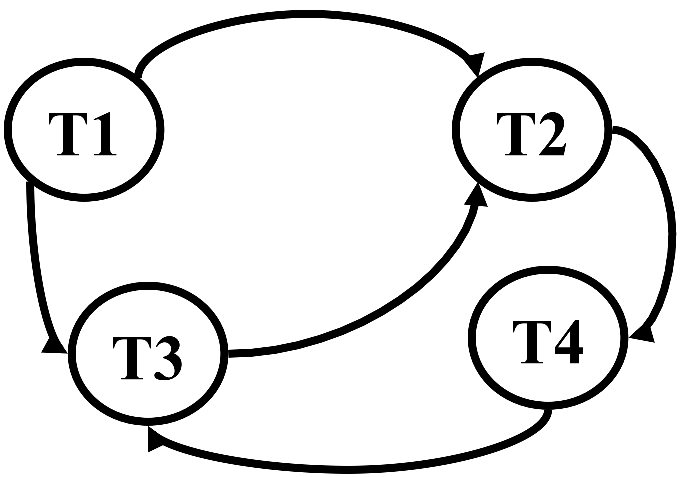
- how often should we run the algo?
- how many transactions are typically involved?

Deadlock handling



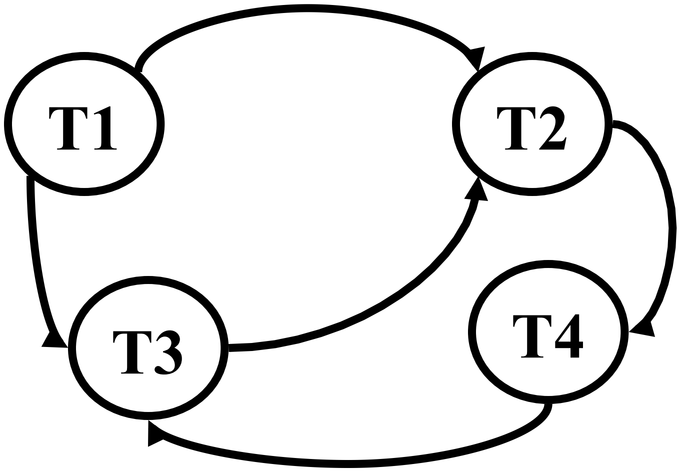
- Q: what to do?

Deadlock handling



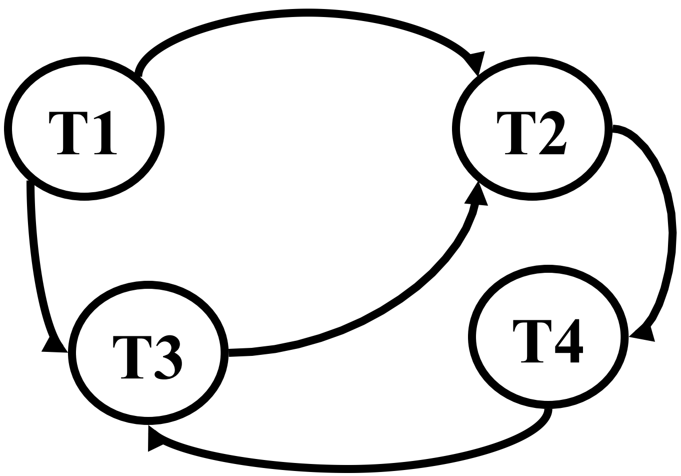
- Q0: what to do?
  - A: select a ‘victim’ & ‘rollback’
- Q1: which/how to choose?

Deadlock handling



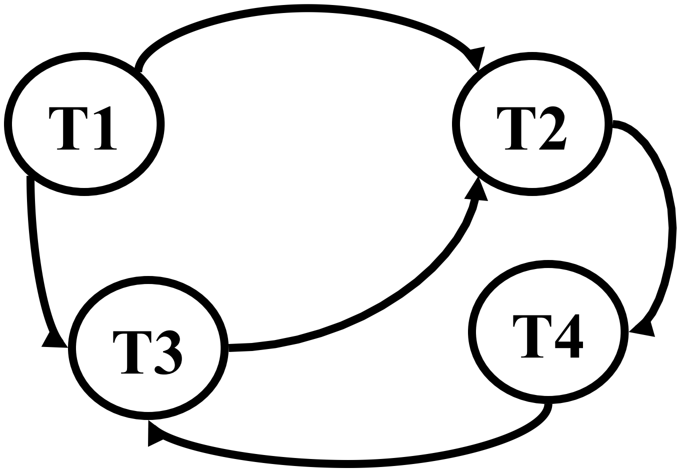
- Q1: which/how to choose?
  - A1.1: by age
  - A1.2: by progress
  - A1.3: by # items locked already...
  - A1.4: by # xacts to rollback
- Q2: How far to rollback?

Deadlock handling



- Q2: How far to rollback?
  - A2.1: completely
  - A2.2: minimally
- Q3: Starvation??

Deadlock handling



- Q3: Starvation??
- A3.1: include #rollbacks in victim selection criterion.

Outline

- Lock Management
- Deadlocks
  - detection
  - **Prevention**
- Specialized Locking

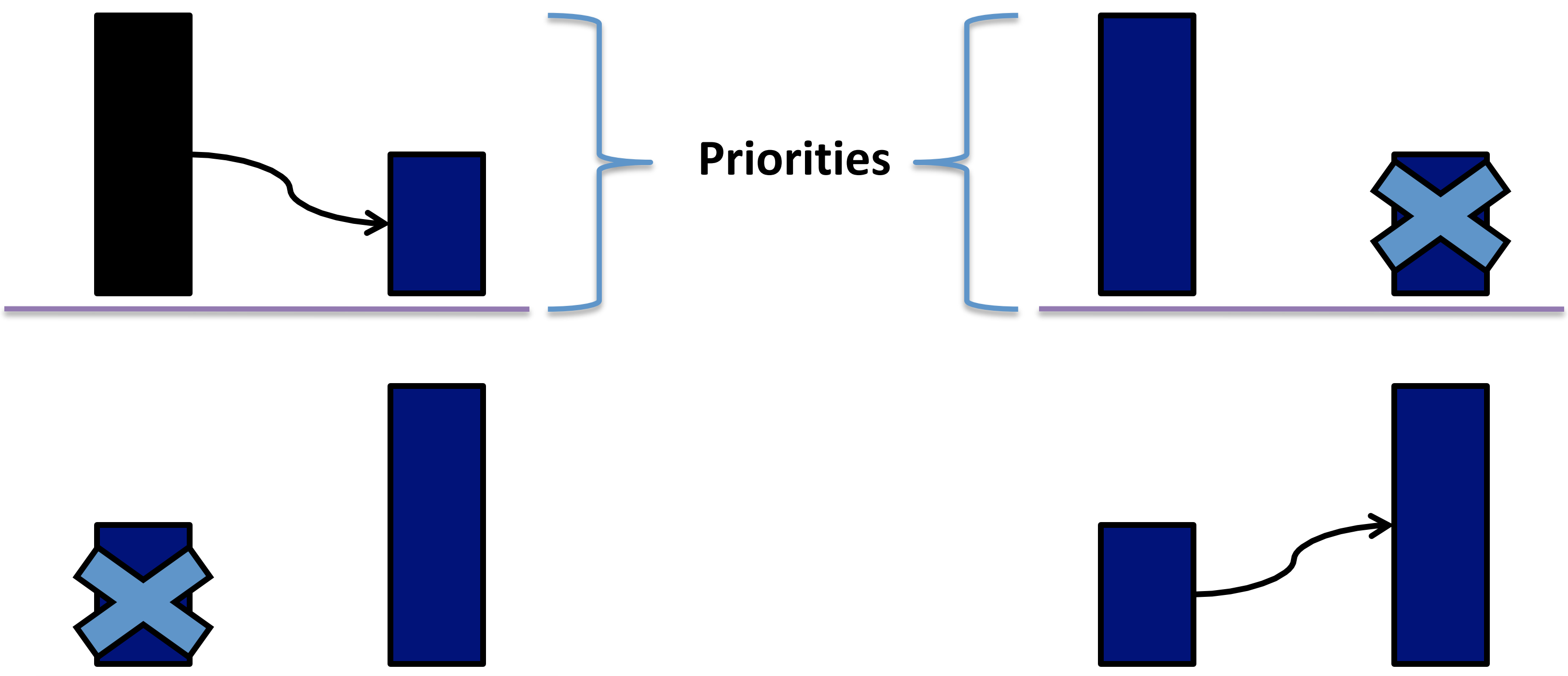
Deadlock Prevention

- Assign priorities based on timestamps (older -> higher priority)
- We only allow 'old-wait-for-young'
- (or only allow 'young-wait-for-old')
- and rollback violators. Specifically:
- Say Ti wants a lock that Tj holds - two policies:
  – Wait-Die: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts (ie., old wait for young)
  – Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits (ie., young wait for old)

Deadlock Prevention

Wait-Die | **Wound-Wait**

Ti wants Tj has | Ti wants Tj has



VirginiaTech

Deadlock Prevention

- Q: Why do these schemes guarantee no deadlocks?
- A:
- Q: When a transaction restarts, what is its (new) priority?
- A:

Deadlock Prevention

- Q: Why do these schemes guarantee no deadlocks?
- A: only one ‘type’ of direction allowed.
- Q: When a transaction restarts, what is its (new) priority?
- A: its original timestamp. -- Why?

SQL statement

- usually, conc. control is transparent to the user, but
- LOCK <table-name> [EXCLUSIVE|SHARED]

Quiz:

- is there a serial schedule (= interleaving) that is not serializable?
- is there a serializable schedule that is not serial?
- can 2PL produce a non-serializable schedule? (assume no deadlocks)

Quiz - cont' d

- is there a serializable schedule that can not be produced by 2PL?
- a xact obeys 2PL - can it be involved in a non-serializable schedule?
- all xacts obey 2PL - can they end up in a deadlock?

Outline

- Lock Management
- Deadlocks
  - detection
  - Prevention
- **Specialized Locking**

Things we will not study

SKIP

- We assumed till now DB objects are fixed and independent---not true in many cases!
- Multi-level locking
  - Lock db or file or pages or record?
- What about locking indexes?
  - E.g. B+-trees
  - Crabbing Algorithm
- What about dynamic databases?
  - ‘phantom’ problem
  - Solution: predicate locking
- Non-locking based Techniques
  - Timestamp based Concurrency Control
- All these are in the textbook though

Transaction Support in SQL-92

recommended

- **SERIALIZABLE** – No phantoms, all reads repeatable, no "dirty" (uncommited) reads.
- REPEATABLE READS – phantoms may happen.
- READ COMMITTED – phantoms and unrepeatable reads may happen
- READ UNCOMMITTED – all of them may happen.

Transaction Support in SQL-92

- SERIALIZABLE : obtains all locks first; plus index locks, plus strict 2PL
- REPEATABLE READS – as above, but no index locks
- READ COMMITTED – as above, but S-locks are released immediately
- READ UNCOMMITTED – as above, but allowing 'dirty reads' (no S-locks)

Transaction Support in SQL-92

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY

- Defaults:
- SERIALIZABLE ← **isolation level**
- READ WRITE ← **access mode**

Conclusions

- 2PL/2PL-C (=Strict 2PL): extremely popular
- Deadlock may still happen
  - detection: wait-for graph
  - prevention: abort some xacts, defensively
- philosophically: concurrency control uses:
  - locks
  - and aborts