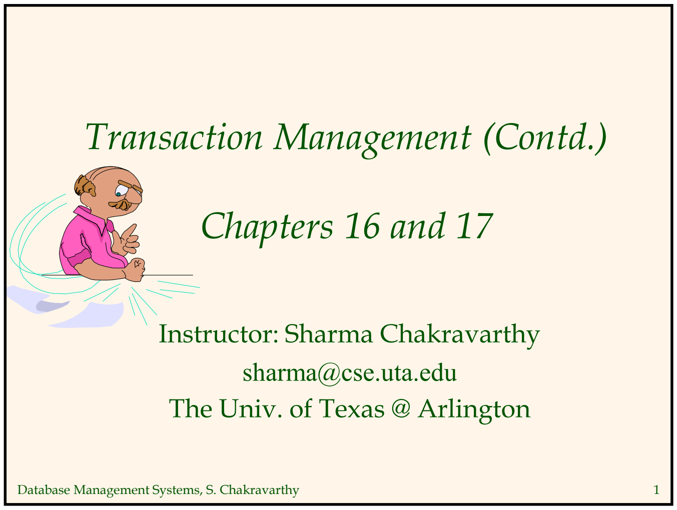
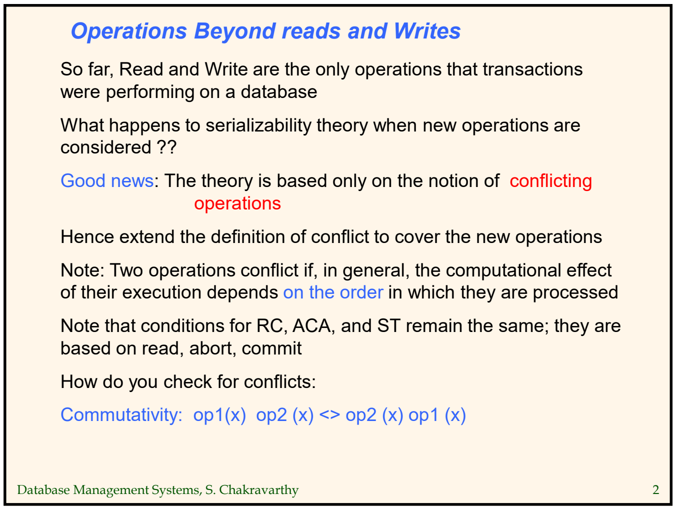


Transaction Management (Contd.)

Chapters 16 and 17

Instructor: Sharma Chakravarthy
sharma@cse.uta.edu
The Univ. of Texas @ Arlington

1



Operations Beyond reads and Writes

So far, Read and Write are the only operations that transactions were performing on a database

What happens to serializability theory when new operations are considered ??

Good news: The theory is based only on the notion of conflicting operations

Hence extend the definition of conflict to cover the new operations

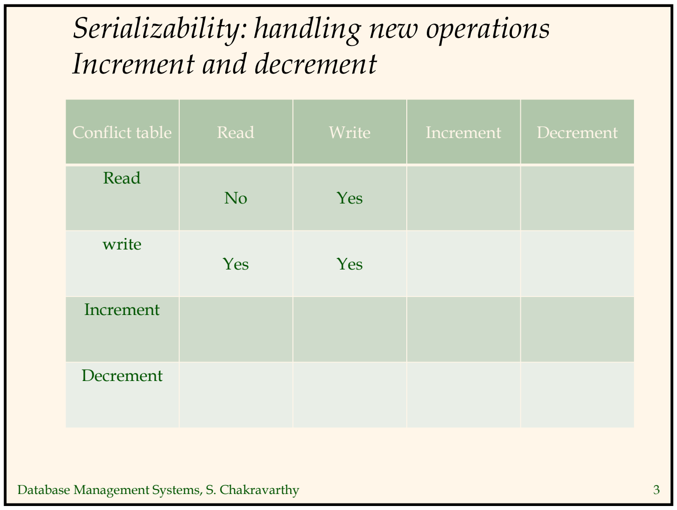
Note: Two operations conflict if, in general, the computational effect of their execution depends on the order in which they are processed

Note that conditions for RC, ACA, and ST remain the same; they are based on read, abort, commit

How do you check for conflicts:

Commutativity: op1(x) op2 (x) <> op2 (x) op1 (x)

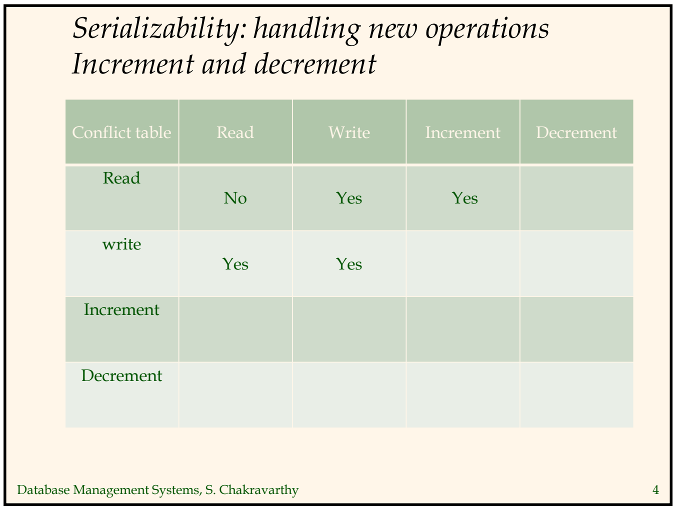
2



Serializability: handling new operations Increment and decrement

Conflict table	Read	Write	Increment	Decrement
Read	No	Yes		
write	Yes	Yes		
Increment				
Decrement				

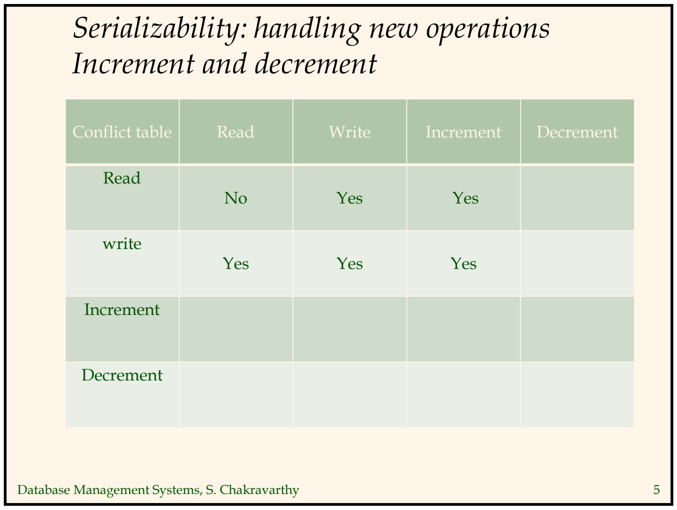
3



Serializability: handling new operations Increment and decrement

Conflict table	Read	Write	Increment	Decrement
Read	No	Yes	Yes	
write	Yes	Yes		
Increment				
Decrement				

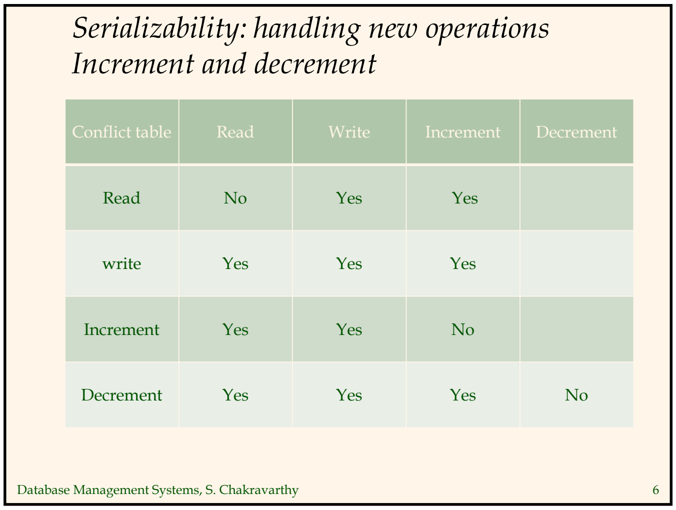
4



Serializability: handling new operations Increment and decrement

Conflict table	Read	Write	Increment	Decrement
Read	No	Yes	Yes	
write	Yes	Yes	Yes	
Increment				
Decrement				

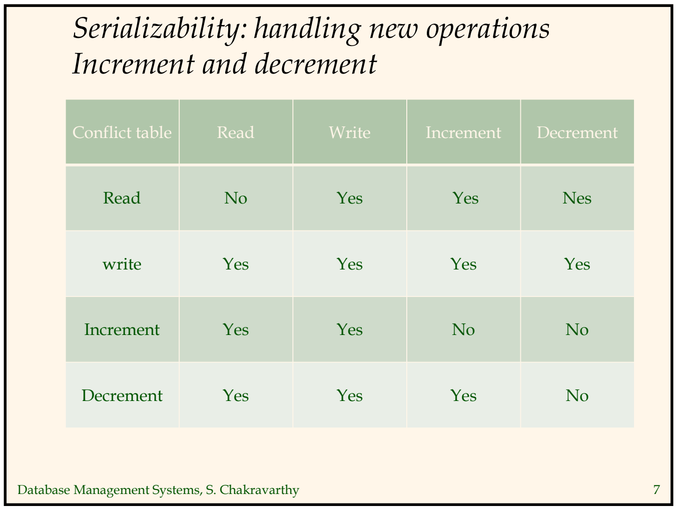
5



Serializability: handling new operations Increment and decrement

Conflict table	Read	Write	Increment	Decrement
Read	No	Yes	Yes	
write	Yes	Yes	Yes	
Increment	Yes	Yes	No	
Decrement	Yes	Yes	Yes	No

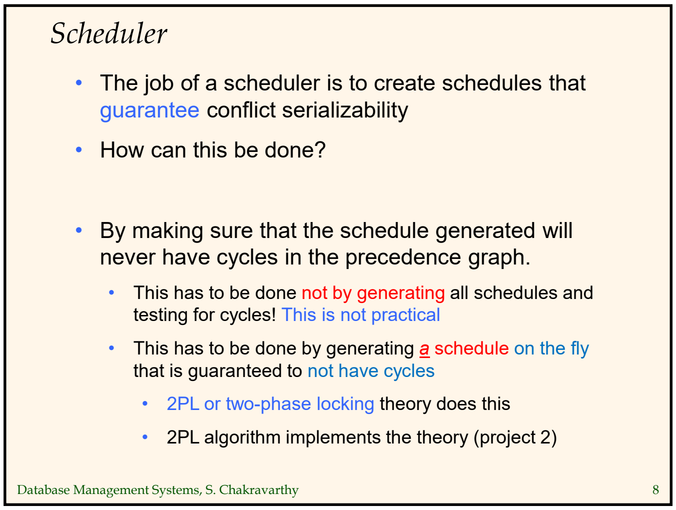
6



Serializability: handling new operations Increment and decrement

Conflict table	Read	Write	Increment	Decrement
Read	No	Yes	Yes	Nes
write	Yes	Yes	Yes	Yes
Increment	Yes	Yes	No	No
Decrement	Yes	Yes	Yes	No

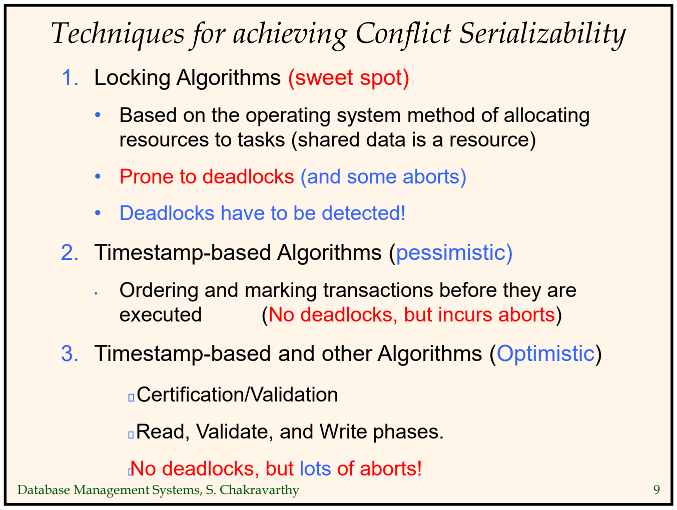
7



Scheduler

- The job of a scheduler is to create schedules that guarantee conflict serializability
- How can this be done?
- By making sure that the schedule generated will never have cycles in the precedence graph.
  - This has to be done not by generating all schedules and testing for cycles! This is not practical
  - This has to be done by generating a schedule on the fly that is guaranteed to not have cycles
    - 2PL or two-phase locking theory does this
    - 2PL algorithm implements the theory (project 2)

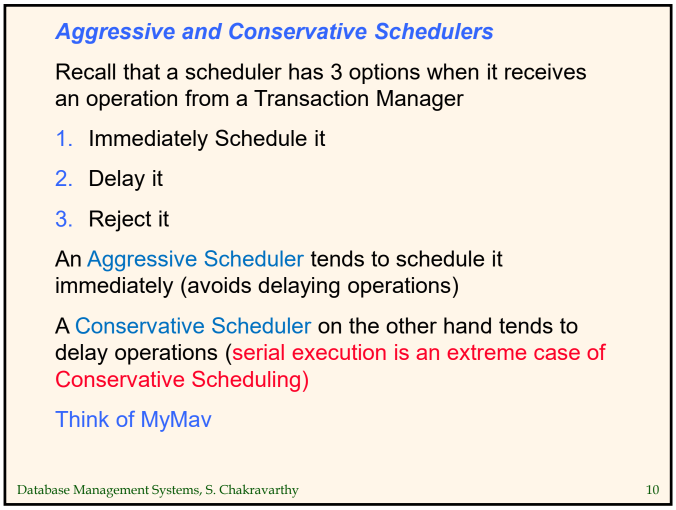
8



Techniques for achieving Conflict Serializability

1. Locking Algorithms (sweet spot)
   - Based on the operating system method of allocating resources to tasks (shared data is a resource)
   - Prone to deadlocks (and some aborts)
   - Deadlocks have to be detected!
2. Timestamp-based Algorithms (pessimistic)
   - Ordering and marking transactions before they are executed (No deadlocks, but incurs aborts)
3. Timestamp-based and other Algorithms (Optimistic)
   - Certification/Validation
   - Read, Validate, and Write phases.
   - No deadlocks, but lots of aborts!

9



Aggressive and Conservative Schedulers

Recall that a scheduler has 3 options when it receives an operation from a Transaction Manager

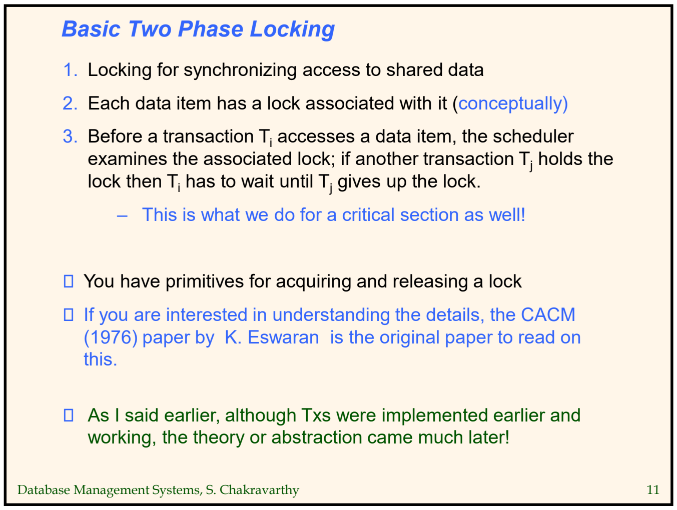
1. Immediately Schedule it
2. Delay it
3. Reject it

An Aggressive Scheduler tends to schedule it immediately (avoids delaying operations)

A Conservative Scheduler on the other hand tends to delay operations (serial execution is an extreme case of Conservative Scheduling)

Think of MyMav

10



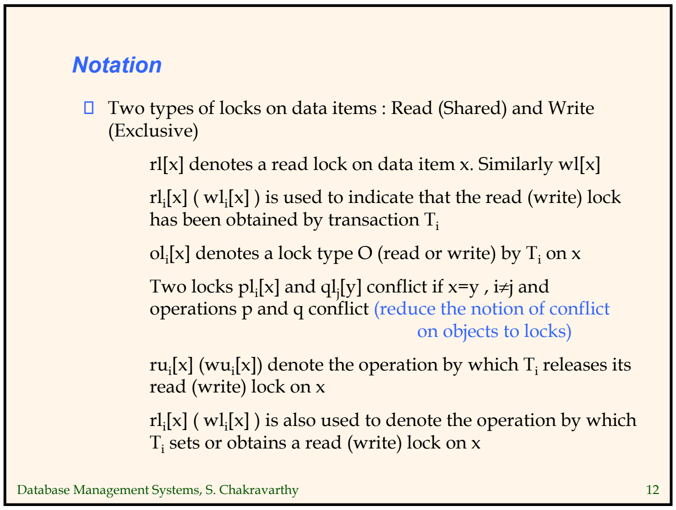
Basic Two Phase Locking

1. Locking for synchronizing access to shared data
2. Each data item has a lock associated with it (conceptually)
3. Before a transaction T_i accesses a data item, the scheduler examines the associated lock; if another transaction T_j holds the lock then T_i has to wait until T_j gives up the lock.
   - This is what we do for a critical section as well!

- You have primitives for acquiring and releasing a lock
- If you are interested in understanding the details, the CACM (1976) paper by K. Eswaran is the original paper to read on this.

- As I said earlier, although Txs were implemented earlier and working, the theory or abstraction came much later!

11



Notation

- Two types of locks on data items : Read (Shared) and Write (Exclusive)

  rl[x] denotes a read lock on data item x. Similarly wl[x]

  $rl_i[x]$ ($wl_i[x]$) is used to indicate that the read (write) lock has been obtained by transaction T_i

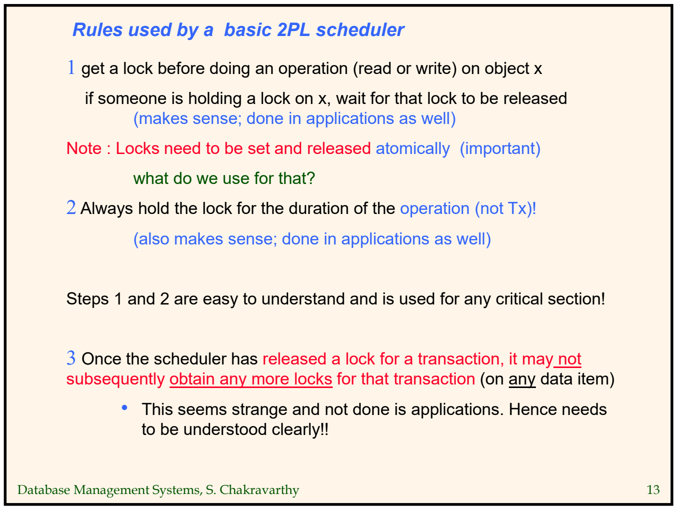
  $ol_i[x]$ denotes a lock type O (read or write) by T_i on x

  Two locks $pl_i[x]$ and $ql_j[y]$ conflict if x=y , i≠j and operations p and q conflict (reduce the notion of conflict on objects to locks)

  $ru_i[x]$ ($wu_i[x]$) denote the operation by which T_i releases its read (write) lock on x

  $rl_i[x]$ ($wl_i[x]$) is also used to denote the operation by which T_i sets or obtains a read (write) lock on x

12



Rules used by a basic 2PL scheduler

1 get a lock before doing an operation (read or write) on object x

if someone is holding a lock on x, wait for that lock to be released (makes sense; done in applications as well)

Note : Locks need to be set and released atomically (important)

what do we use for that?

2 Always hold the lock for the duration of the operation (not Tx)!

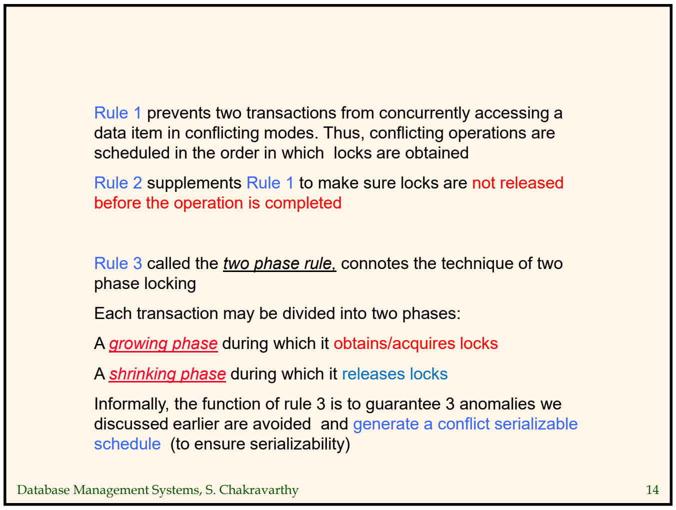
(also makes sense; done in applications as well)

Steps 1 and 2 are easy to understand and is used for any critical section!

3 Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item)

- This seems strange and not done is applications. Hence needs to be understood clearly!!

13



Rule 1 prevents two transactions from concurrently accessing a data item in conflicting modes. Thus, conflicting operations are scheduled in the order in which locks are obtained

Rule 2 supplements Rule 1 to make sure locks are not released before the operation is completed

Rule 3 called the *two phase rule,* connotes the technique of two phase locking

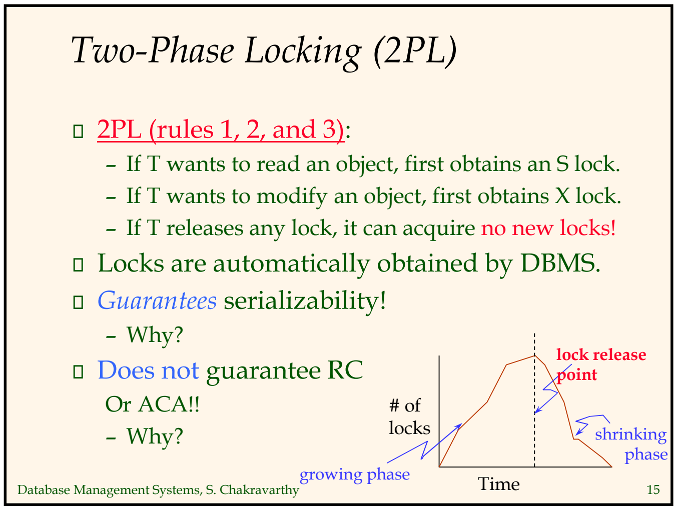
Each transaction may be divided into two phases:

A *growing phase* during which it obtains/acquires locks

A *shrinking phase* during which it releases locks

Informally, the function of rule 3 is to guarantee 3 anomalies we discussed earlier are avoided and generate a conflict serializable schedule (to ensure serializability)

14

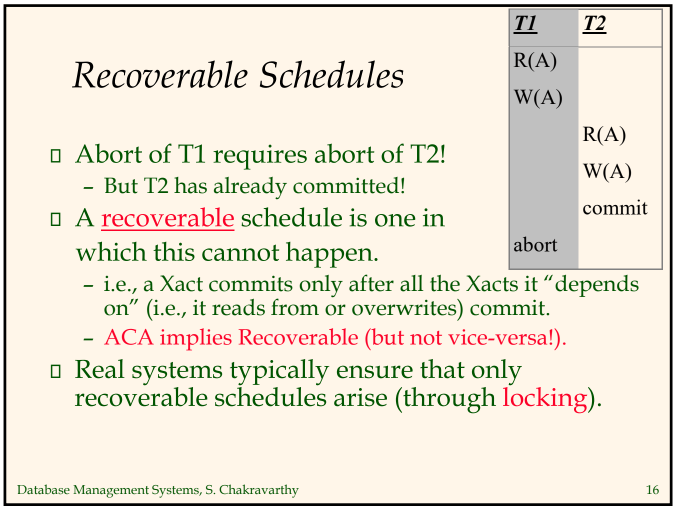


Two-Phase Locking (2PL)

- 2PL (rules 1, 2, and 3):
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - If T releases any lock, it can acquire no new locks!
- Locks are automatically obtained by DBMS.
- *Guarantees* serializability!
  - Why?
- Does not guarantee RC
  - Or ACA!!
  - Why?

lock release point
# of locks
shrinking phase
growing phase
Time

15

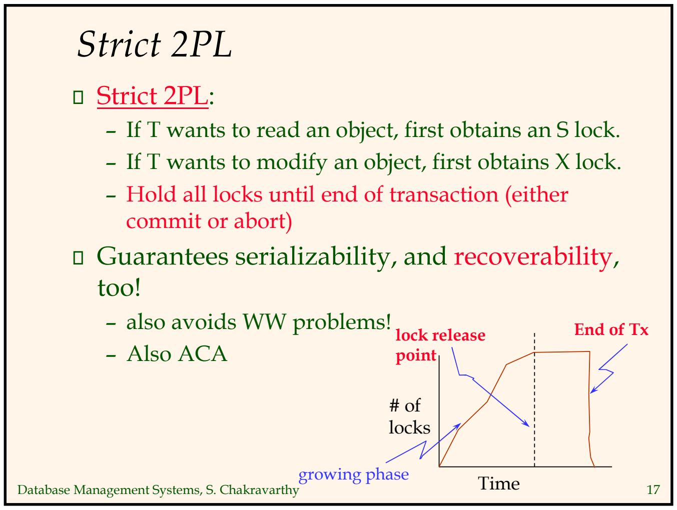


Recoverable Schedules

T1	***T2***
R(A)	
W(A)	
	R(A)
	W(A)
	commit
abort	

- Abort of T1 requires abort of T2!
  - But T2 has already committed!
- A recoverable schedule is one in which this cannot happen.
  - i.e., a Xact commits only after all the Xacts it "depends on" (i.e., it reads from or overwrites) commit.
  - ACA implies Recoverable (but not vice-versa!).
- Real systems typically ensure that only recoverable schedules arise (through locking).

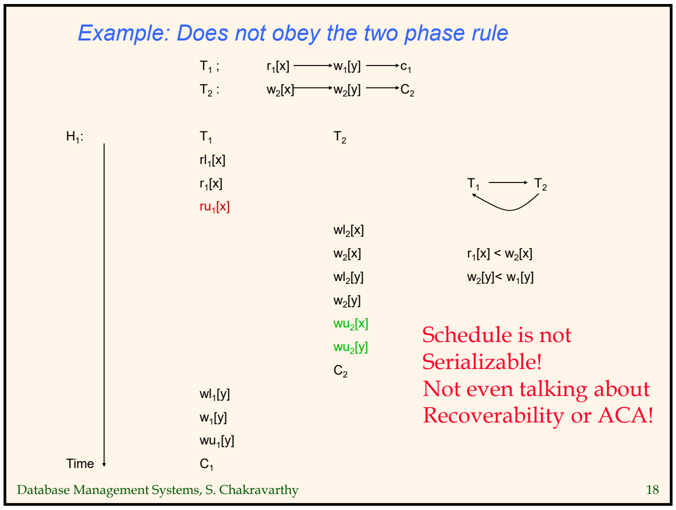
16



Strict 2PL

- Strict 2PL:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - Hold all locks until end of transaction (either commit or abort)
- Guarantees serializability, and recoverability, too!
  - also avoids WW problems!
  - Also ACA

lock release point
End of Tx
# of locks
growing phase
Time



Example: Does not obey the two phase rule

T_1 ; $r_1[x] \longrightarrow w_1[y] \longrightarrow c_1$

T_2 : $w_2[x] \longrightarrow w_2[y] \longrightarrow C_2$

H_1:

T_1	T_2
$rl_1[x]$	
$r_1[x]$	
$ru_1[x]$	
	$wl_2[x]$
	$w_2[x]$
	$wl_2[y]$
	$w_2[y]$
	$wu_2[x]$
	$wu_2[y]$
	C_2
$wl_1[y]$	
$w_1[y]$	
$wu_1[y]$	
C_1	

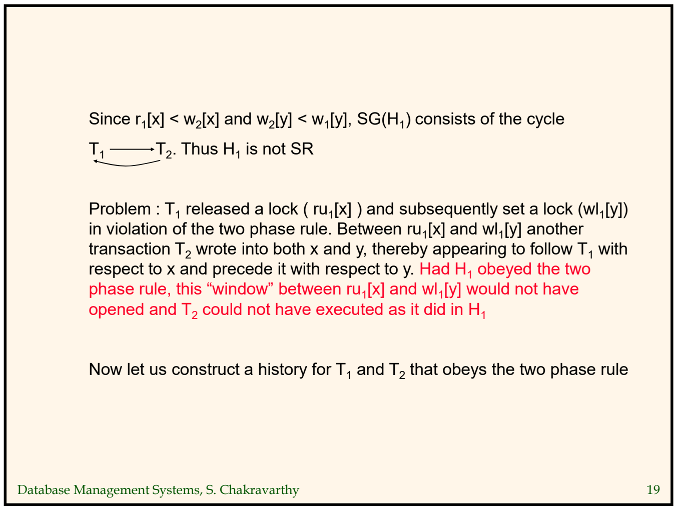
Time

$$\begin{tikzcd} T_1 \arrow[r] & T_2 \arrow[l, bend left] \end{tikzcd}$$

$r_1[x] < w_2[x]$

$w_2[y]< w_1[y]$

Schedule is not Serializable!
Not even talking about Recoverability or ACA!

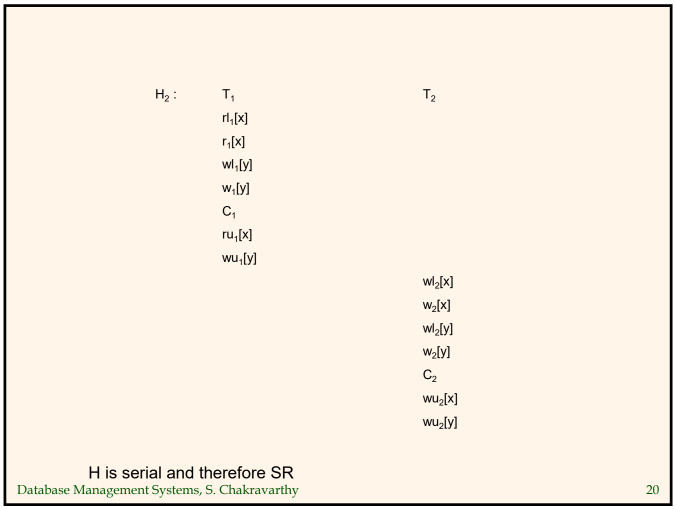


Since $r_1[x] < w_2[x]$ and $w_2[y] < w_1[y]$, SG(H_1) consists of the cycle

$\begin{tikzcd} T_1 \arrow[r] & T_2 \arrow[l, bend left] \end{tikzcd}$. Thus H_1 is not SR

Problem : T_1 released a lock ($ru_1[x]$) and subsequently set a lock ($wl_1[y]$) in violation of the two phase rule. Between $ru_1[x]$ and $wl_1[y]$ another transaction T_2 wrote into both x and y, thereby appearing to follow T_1 with respect to x and precede it with respect to y. Had H_1 obeyed the two phase rule, this "window" between $ru_1[x]$ and $wl_1[y]$ would not have opened and T_2 could not have executed as it did in H_1

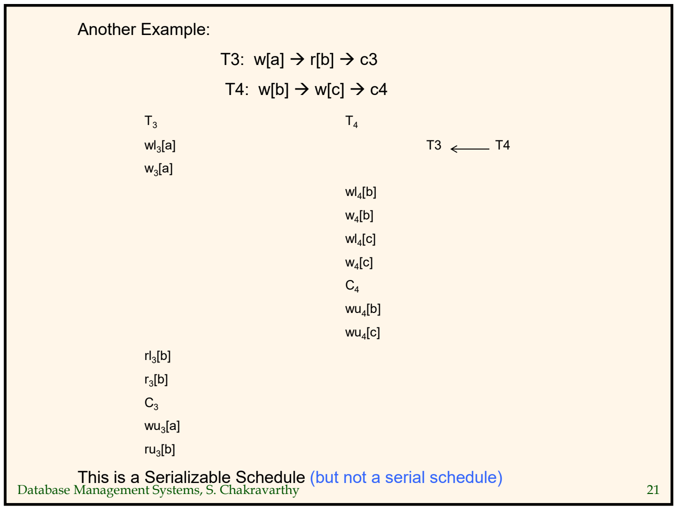
Now let us construct a history for T_1 and T_2 that obeys the two phase rule



H_2 :

T_1	T_2
$rl_1[x]$	
$r_1[x]$	
$wl_1[y]$	
$w_1[y]$	
C_1	
$ru_1[x]$	
$wu_1[y]$	
	$wl_2[x]$
	$w_2[x]$
	$wl_2[y]$
	$w_2[y]$
	C_2
	$wu_2[x]$
	$wu_2[y]$

H is serial and therefore SR



Another Example:

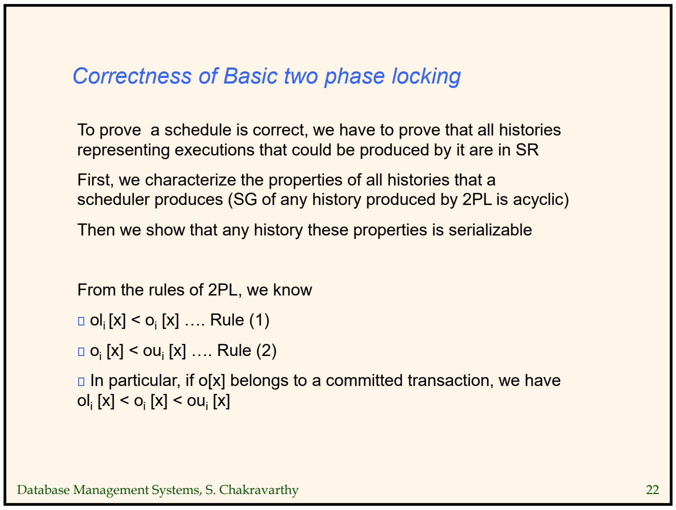
T3: w[a] → r[b] → c3

T4: w[b] → w[c] → c4

T_3	T_4
$wl_3[a]$	
$w_3[a]$	
	$wl_4[b]$
	$w_4[b]$
	$wl_4[c]$
	$w_4[c]$
	C_4
	$wu_4[b]$
	$wu_4[c]$
$rl_3[b]$	
$r_3[b]$	
C_3	
$wu_3[a]$	
$ru_3[b]$	

$T3 \longleftarrow T4$

This is a Serializable Schedule (but not a serial schedule)



Correctness of Basic two phase locking

To prove a schedule is correct, we have to prove that all histories representing executions that could be produced by it are in SR

First, we characterize the properties of all histories that a scheduler produces (SG of any history produced by 2PL is acyclic)

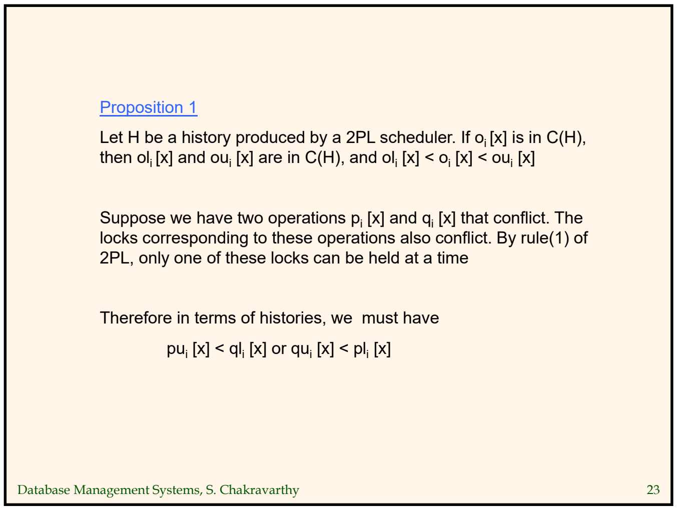
Then we show that any history these properties is serializable

From the rules of 2PL, we know

□ ol_i [x] < o_i [x] …. Rule (1)

□ o_i [x] < ou_i [x] …. Rule (2)

□ In particular, if o[x] belongs to a committed transaction, we have ol_i [x] < o_i [x] < ou_i [x]



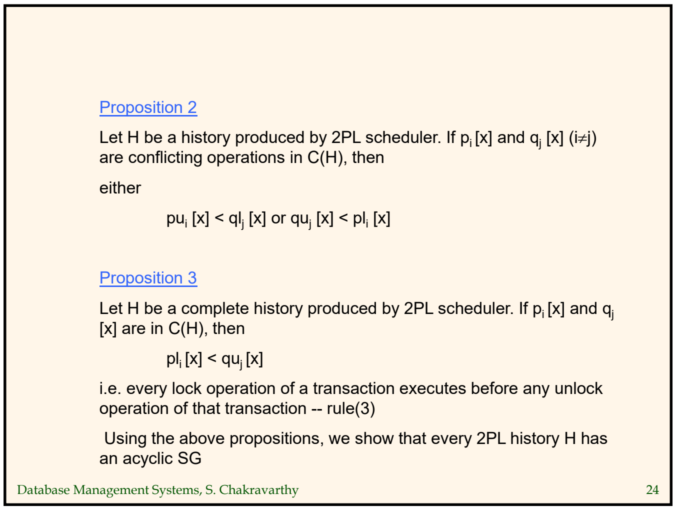
Proposition 1

Let H be a history produced by a 2PL scheduler. If o_i [x] is in C(H), then ol_i [x] and ou_i [x] are in C(H), and ol_i [x] < o_i [x] < ou_i [x]

Suppose we have two operations p_i [x] and q_i [x] that conflict. The locks corresponding to these operations also conflict. By rule(1) of 2PL, only one of these locks can be held at a time

Therefore in terms of histories, we must have

pu_i [x] < ql_i [x] or qu_i [x] < pl_i [x]



Proposition 2

Let H be a history produced by 2PL scheduler. If p_i [x] and q_j [x] (i≠j) are conflicting operations in C(H), then

either

pu_i [x] < ql_j [x] or qu_j [x] < pl_i [x]

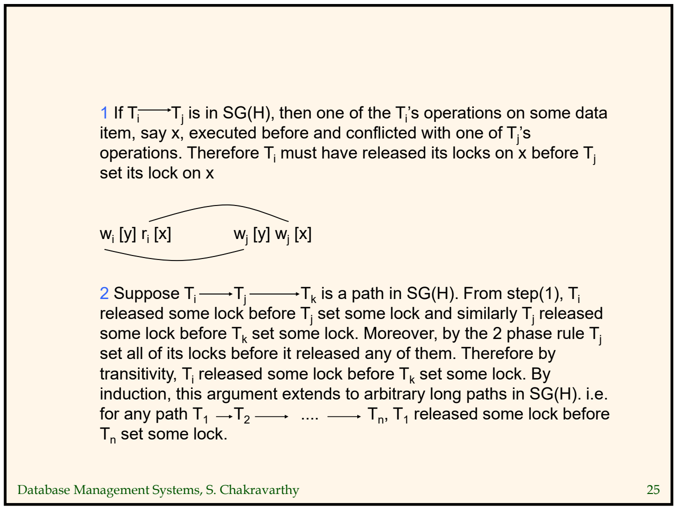
Proposition 3

Let H be a complete history produced by 2PL scheduler. If p_i [x] and q_j [x] are in C(H), then

pl_i [x] < qu_j [x]

i.e. every lock operation of a transaction executes before any unlock operation of that transaction -- rule(3)

Using the above propositions, we show that every 2PL history H has an acyclic SG

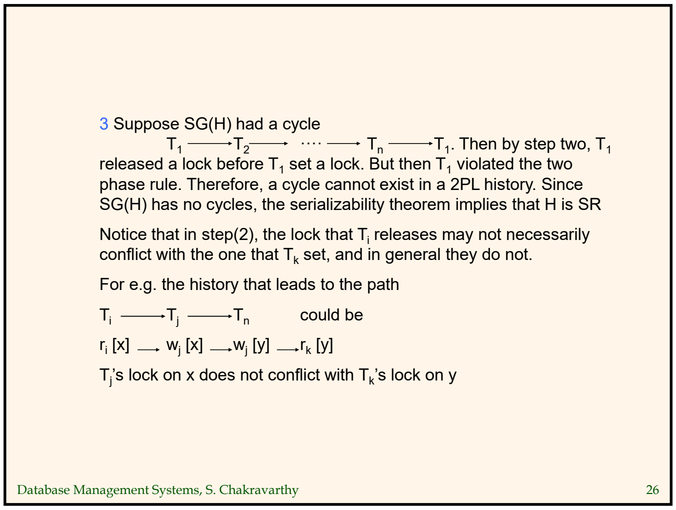


1 If $T_i \longrightarrow T_j$ is in SG(H), then one of the T_i's operations on some data item, say x, executed before and conflicted with one of T_j's operations. Therefore T_i must have released its locks on x before T_j set its lock on x

$w_i\,[y]\; r_i\,[x] \qquad w_j\,[y]\; w_j\,[x]$

2 Suppose $T_i \longrightarrow T_j \longrightarrow T_k$ is a path in SG(H). From step(1), T_i released some lock before T_j set some lock and similarly T_j released some lock before T_k set some lock. Moreover, by the 2 phase rule T_j set all of its locks before it released any of them. Therefore by transitivity, T_i released some lock before T_k set some lock. By induction, this argument extends to arbitrary long paths in SG(H). i.e. for any path $T_1 \rightarrow T_2 \longrightarrow \ldots \longrightarrow T_n$, T_1 released some lock before T_n set some lock.

25



3 Suppose SG(H) had a cycle $T_1 \longrightarrow T_2 \longrightarrow \cdots \longrightarrow T_n \longrightarrow T_1$. Then by step two, T_1 released a lock before T_1 set a lock. But then T_1 violated the two phase rule. Therefore, a cycle cannot exist in a 2PL history. Since SG(H) has no cycles, the serializability theorem implies that H is SR

Notice that in step(2), the lock that T_i releases may not necessarily conflict with the one that T_k set, and in general they do not.

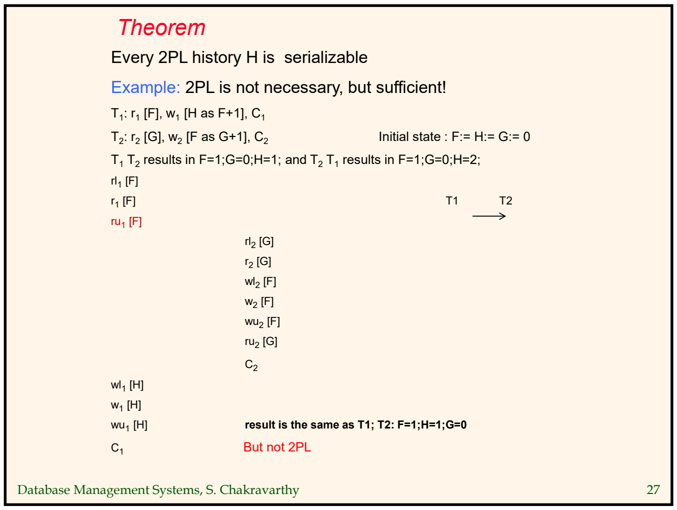
For e.g. the history that leads to the path

$T_i \longrightarrow T_j \longrightarrow T_n$ could be

$r_i\,[x] \longrightarrow w_j\,[x] \longrightarrow w_j\,[y] \longrightarrow r_k\,[y]$

T_j's lock on x does not conflict with T_k's lock on y

26



Theorem

Every 2PL history H is serializable

Example: 2PL is not necessary, but sufficient!

T_1: r_1 [F], w_1 [H as F+1], C_1

T_2: r_2 [G], w_2 [F as G+1], C_2 Initial state : F:= H:= G:= 0

T_1 T_2 results in F=1;G=0;H=1; and T_2 T_1 results in F=1;G=0;H=2;

rl_1 [F]

r_1 [F] $\quad T1 \longrightarrow T2$

ru_1 [F]

rl_2 [G]

r_2 [G]

wl_2 [F]

w_2 [F]

wu_2 [F]

ru_2 [G]

C_2

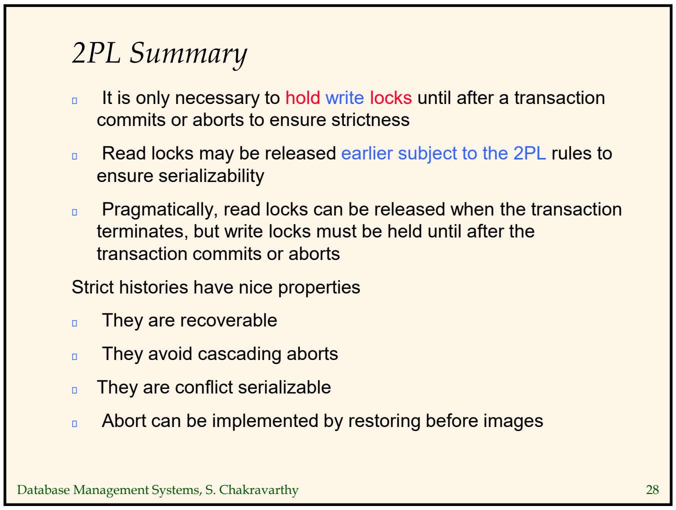
wl_1 [H]

w_1 [H]

wu_1 [H] **result is the same as T1; T2: F=1;H=1;G=0**

C_1 But not 2PL

27



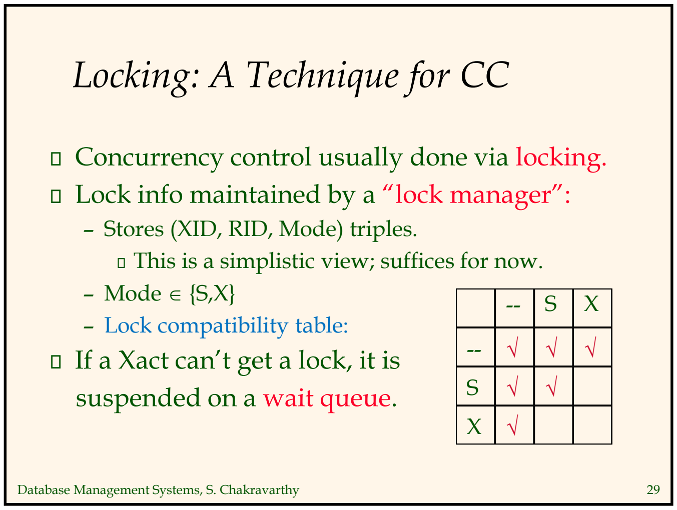
2PL Summary

- It is only necessary to hold write locks until after a transaction commits or aborts to ensure strictness
- Read locks may be released earlier subject to the 2PL rules to ensure serializability
- Pragmatically, read locks can be released when the transaction terminates, but write locks must be held until after the transaction commits or aborts

Strict histories have nice properties

- They are recoverable
- They avoid cascading aborts
- They are conflict serializable
- Abort can be implemented by restoring before images

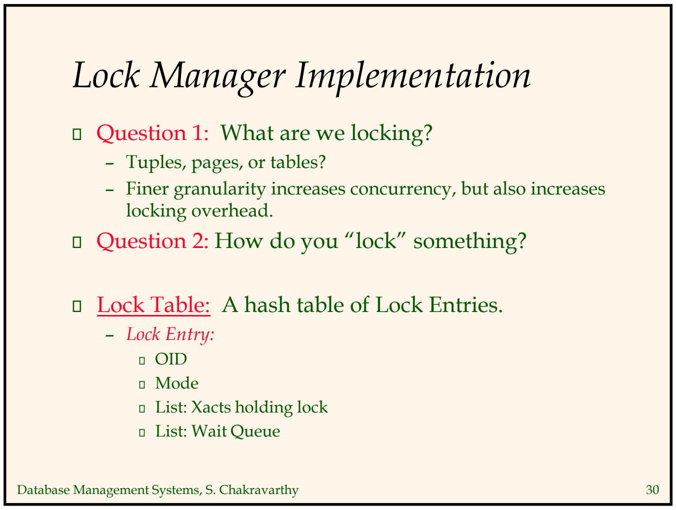
28



Locking: A Technique for CC

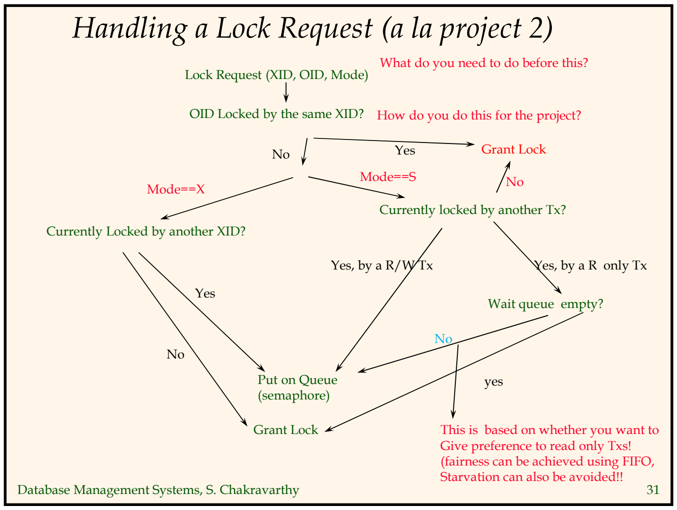
- Concurrency control usually done via locking.
- Lock info maintained by a “lock manager”:
  - Stores (XID, RID, Mode) triples.
    - This is a simplistic view; suffices for now.
  - Mode ∈ {S,X}
  - Lock compatibility table:
- If a Xact can’t get a lock, it is suspended on a wait queue.

	--	S	X
--	√	√	√
S	√	√	
X	√		



Lock Manager Implementation

- Question 1: What are we locking?
  - Tuples, pages, or tables?
  - Finer granularity increases concurrency, but also increases locking overhead.
- Question 2: How do you “lock” something?
- Lock Table: A hash table of Lock Entries.
  - *Lock Entry:*
    - OID
    - Mode
    - List: Xacts holding lock
    - List: Wait Queue



Handling a Lock Request (a la project 2)

Lock Request (XID, OID, Mode)

What do you need to do before this?

OID Locked by the same XID?

How do you do this for the project?

Yes

Grant Lock

No

Mode==X

Mode==S

No

Currently locked by another Tx?

Currently Locked by another XID?

Yes, by a R/W Tx

Yes, by a R only Tx

Yes

Wait queue empty?

No

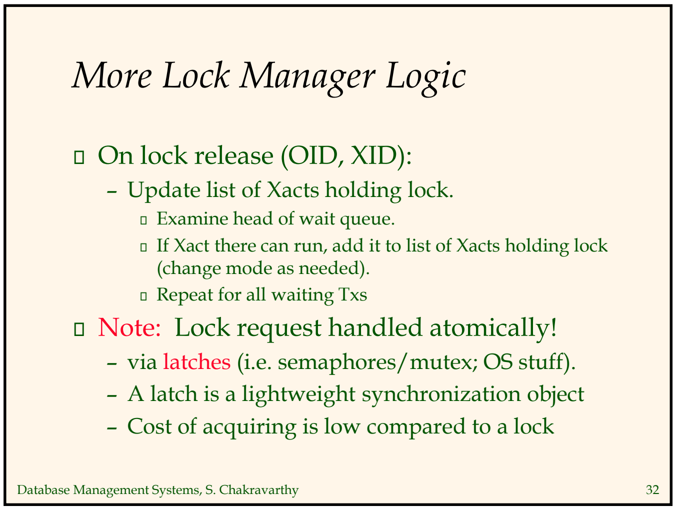
No

Put on Queue
(semaphore)

yes

Grant Lock

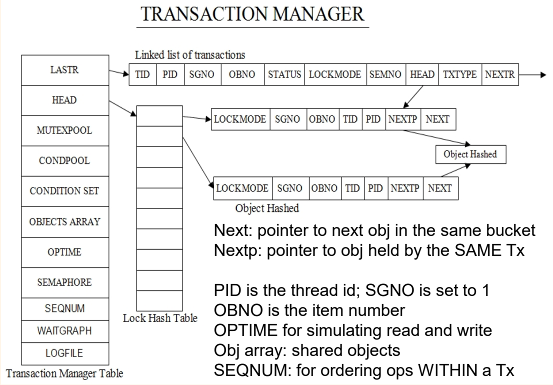
This is based on whether you want to Give preference to read only Txs! (fairness can be achieved using FIFO, Starvation can also be avoided!!



More Lock Manager Logic

- On lock release (OID, XID):
  - Update list of Xacts holding lock.
    - Examine head of wait queue.
    - If Xact there can run, add it to list of Xacts holding lock (change mode as needed).
    - Repeat for all waiting Txs
- Note: Lock request handled atomically!
  - via latches (i.e. semaphores/mutex; OS stuff).
  - A latch is a lightweight synchronization object
  - Cost of acquiring is low compared to a lock

Tx Mgr used for project 2



Next: pointer to next obj in the same bucket
Nextp: pointer to obj held by the SAME Tx

PID is the thread id; SGNO is set to 1
OBNO is the item number
OPTIME for simulating read and write
Obj array: shared objects
SEQNUM: for ordering ops WITHIN a Tx

Lock Upgrades

- Think about this scenario:
  - T1 locks A in S mode, T2 requests X lock on A, T3 requests S lock on A. *should we allow T3 or make it wait?*
  - *S1(A), X2(A), S3(A)*
  - *Fairness vs. Starvation vs. deadlock! (read only vs. read/write Txs)*
  - *If T3 is given the lock*
    - *It is NOT fair to T2; can lead to starvation of T2*
    - *It will ALSO prevent upgrade for T1. may lead to deadlock*
    - *If we know T1 is read only, previous situation does not happen. Only fairness is the issue*

- In contrast:
  - T1 locks A in S mode, T2 requests X lock on A, T1 requests X lock on A. *What should we do?*
  - *S1(A), X2(A), X1(A)*
  - *If T1 waits, it may never get a chance to proceed! (waiting on itself)*
  - *Deadlock situation*
  - *Hence should allow upgrade!*

Lock Upgrades

- Should we allow such upgrades to supersede lock requests?
  - Yes!!

- Of course, there are genuine deadlocks when 2Pl is followed
  - S1(A), W2(B), W1(B), W2(A)
  - Classic deadlock situation

note: in project 2, due to race conditions, this may not happen

if T1 does ALL its operations and commits, T2 can proceed. that is essentially a serial execution

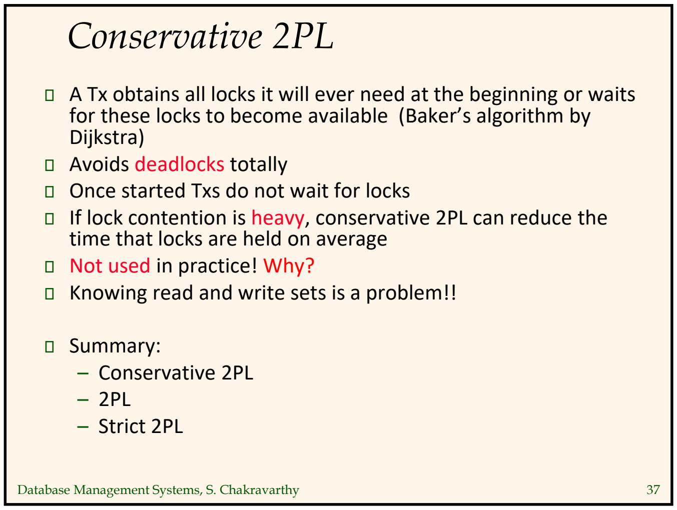
We know serial executions can never result in a deadlock!

Lock upgrades and Deadlocks

- Lock downgrade approach
  - Acquire locks in x mode first
  - Downgrade to S mode when we know that the object is not going to be modified
  - E.g., a tuple does not satisfy the where condition
  - CC is reduced as we are taking the lock initially in X mode
  - 2PL can be modified to accommodate this
  - Commercial systems use this as throughput is increased and deadlocks are avoided!

- Deadlock can occur even without upgrades:
  - X1(A), X2(B), S1(B), S2(A) standard deadlock scenario

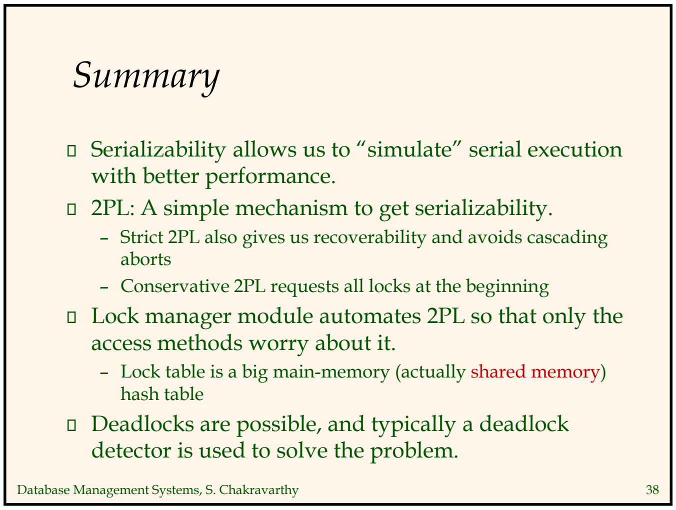
- How do we deal with deadlocks?
  - Deadlock detection (wait for graphs)
  - Deadlock prevention (wound-wait and wait-die)
    - Aborts Txs to prevent deadlocks (and release resources)



Conservative 2PL

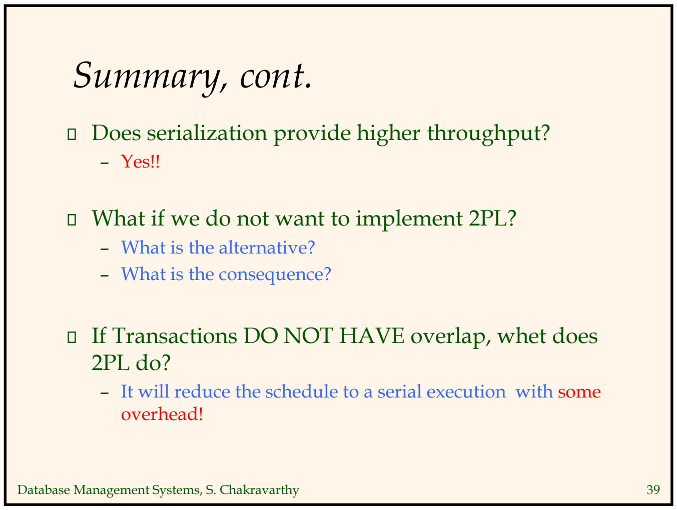
- A Tx obtains all locks it will ever need at the beginning or waits for these locks to become available (Baker's algorithm by Dijkstra)
- Avoids deadlocks totally
- Once started Txs do not wait for locks
- If lock contention is heavy, conservative 2PL can reduce the time that locks are held on average
- Not used in practice! Why?
- Knowing read and write sets is a problem!!

- Summary:
  - Conservative 2PL
  - 2PL
  - Strict 2PL



Summary

- Serializability allows us to "simulate" serial execution with better performance.
- 2PL: A simple mechanism to get serializability.
  - Strict 2PL also gives us recoverability and avoids cascading aborts
  - Conservative 2PL requests all locks at the beginning
- Lock manager module automates 2PL so that only the access methods worry about it.
  - Lock table is a big main-memory (actually shared memory) hash table
- Deadlocks are possible, and typically a deadlock detector is used to solve the problem.

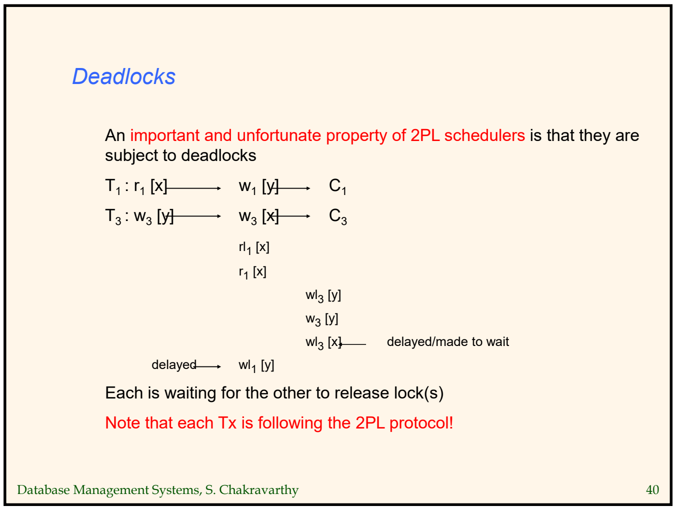


Summary, cont.

- Does serialization provide higher throughput?
  - Yes!!

- What if we do not want to implement 2PL?
  - What is the alternative?
  - What is the consequence?

- If Transactions DO NOT HAVE overlap, whet does 2PL do?
  - It will reduce the schedule to a serial execution with some overhead!



Deadlocks

An important and unfortunate property of 2PL schedulers is that they are subject to deadlocks

$T_1 : r_1 [x] \longrightarrow w_1 [y] \longrightarrow C_1$

$T_3 : w_3 [y] \longrightarrow w_3 [x] \longrightarrow C_3$

$rl_1 [x]$

$r_1 [x]$

$wl_3 [y]$

$w_3 [y]$

$wl_3 [x] \longleftarrow$ delayed/made to wait

delayed $\longrightarrow wl_1 [y]$

Each is waiting for the other to release lock(s)

Note that each Tx is following the 2PL protocol!

Before either of these two processes can proceed, one must release a resource that the other needs to proceed

This situation can also arise when transactions try to escalate read locks into write locks (also known as lock conversion or lock upgrade)

T_4: r_4 [x] → w_4 [x] → C_4

T_5: r_5 [x] → w_5 [x] → C_5

rl_4 [x]

r_4 [x]

rl_5 [x]

r_5 [x]

conflicts with rl_5 [x] → wl_4 [x]

wl_5 [x] ← conflicts with rl_4 [x]

This situation arises when a transaction scans a large number of data items and then decides to update specific data items. If it sets a read lock during the scanning phase which it then tries to strengthen into a write lock during update.

Dealing with Deadlocks

1. Detection -- needs a representation
   - □ Timeout
   - □ Detecting Cycles
2. Prevention (at run time)
   - □ WAIT-DIE
   - □ WOUND-WAIT methods
3. Avoidance
   - □ allocating all the resources required by a transaction first

Representation

- □ Wait-for graph (WFG)
- □ resource Allocation graph

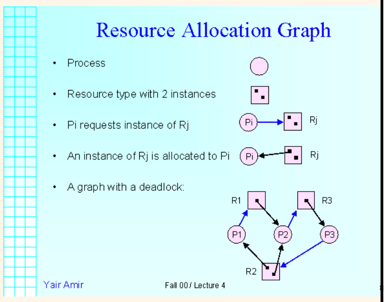
Representation

□ Wait-for Graph (WFG)

The scheduler maintains a WFG. Nodes represent transactions. There is an edge T_i T_j, iff transaction T_i is waiting for transaction T_j to release some lock (some resource)

□ Allocation Graph

A graph consisting of transactions and granules (of data). Arcs (edges) represent lock(s) requested on granules by transactions for specific operations



Resource Allocation Graph

- Process
- Resource type with 2 instances
- Pi requests instance of Rj
- An instance of Rj is allocated to Pi
- A graph with a deadlock:

Yair Amir Fall 00 / Lecture 4

Deadlock Detection Strategies

□ Timeout

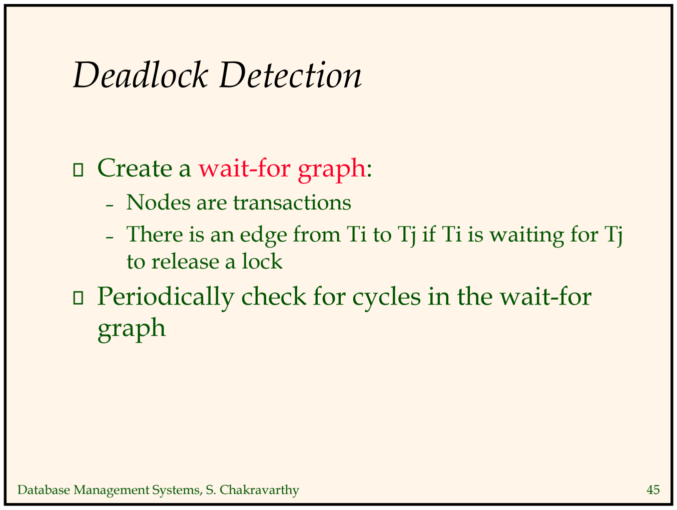
Scheduler is waiting too long for a lock. Then a deadlock situation is assumed and the transaction is aborted

May end up aborting transactions that are not involved in deadlock, but are long. Involves performance penalty; has no bearing on correctness.

If a long timeout is used to overcome the above, the scheduler is likely to abort transactions that are likely (actually) to be involved in deadlock

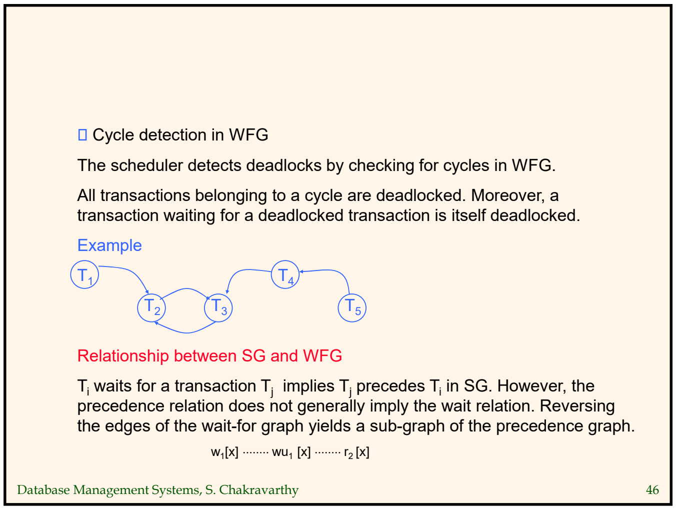
Also, the transaction involved in deadlock loses time as a consequence

Hence, the timeout period is a parameter that needs to be tuned. This activity is tricky, but manageable, as evidenced by its use in several commercial products (e.g. Tandem)



Deadlock Detection

- 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 wait-for graph



- Cycle detection in WFG

The scheduler detects deadlocks by checking for cycles in WFG.

All transactions belonging to a cycle are deadlocked. Moreover, a transaction waiting for a deadlocked transaction is itself deadlocked.

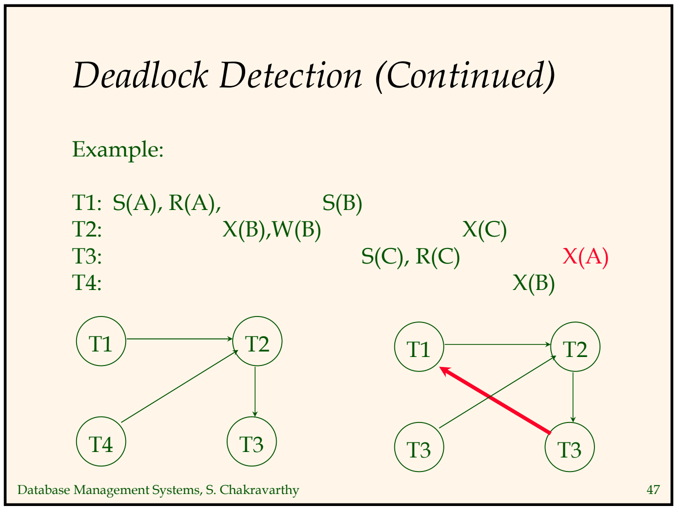
Example

T_1
T_2
T_3
T_4
T_5

Relationship between SG and WFG

T_i waits for a transaction T_j implies T_j precedes T_i in SG. However, the precedence relation does not generally imply the wait relation. Reversing the edges of the wait-for graph yields a sub-graph of the precedence graph.

$w_1[x]$ ········ wu_1 $[x]$ ········ r_2 $[x]$

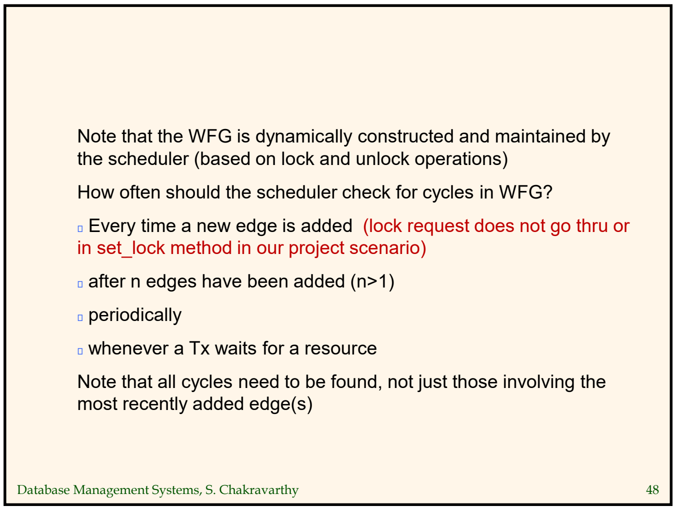


Deadlock Detection (Continued)

Example:

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

T1
T2
T4
T3
T1
T2
T3
T3

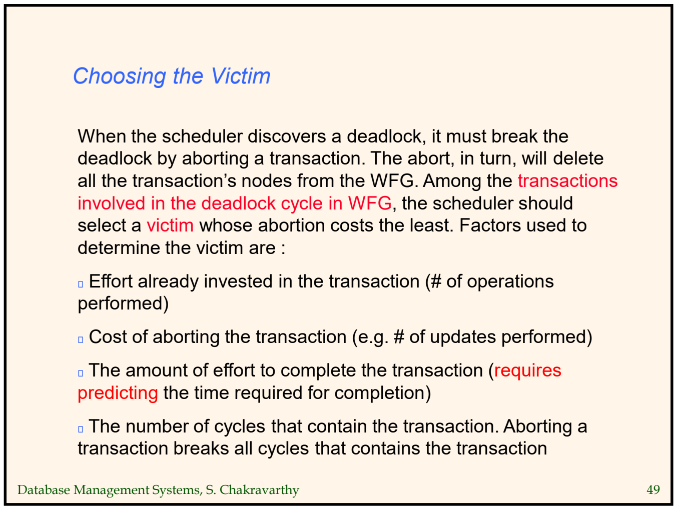


Note that the WFG is dynamically constructed and maintained by the scheduler (based on lock and unlock operations)

How often should the scheduler check for cycles in WFG?

- Every time a new edge is added (lock request does not go thru or in set_lock method in our project scenario)
- after n edges have been added (n>1)
- periodically
- whenever a Tx waits for a resource

Note that all cycles need to be found, not just those involving the most recently added edge(s)

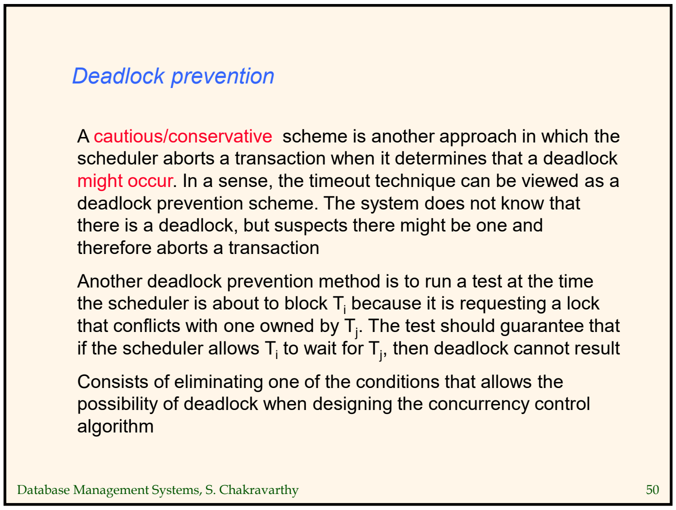


Choosing the Victim

When the scheduler discovers a deadlock, it must break the deadlock by aborting a transaction. The abort, in turn, will delete all the transaction's nodes from the WFG. Among the transactions involved in the deadlock cycle in WFG, the scheduler should select a victim whose abortion costs the least. Factors used to determine the victim are :

- Effort already invested in the transaction (# of operations performed)
- Cost of aborting the transaction (e.g. # of updates performed)
- The amount of effort to complete the transaction (requires predicting the time required for completion)
- The number of cycles that contain the transaction. Aborting a transaction breaks all cycles that contains the transaction

49



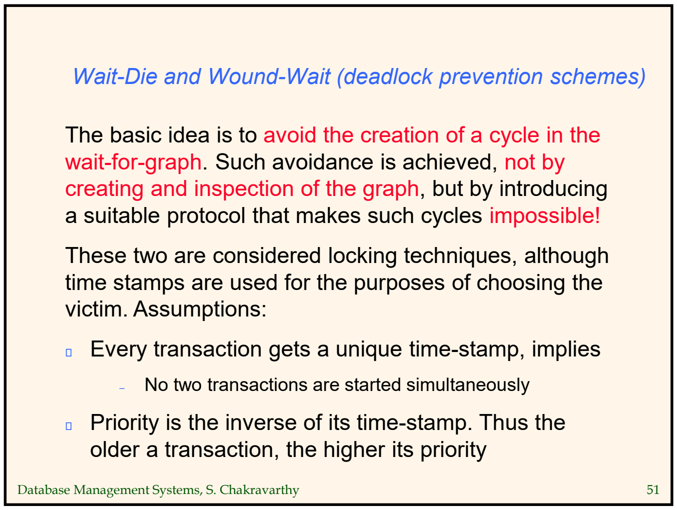
Deadlock prevention

A cautious/conservative scheme is another approach in which the scheduler aborts a transaction when it determines that a deadlock might occur. In a sense, the timeout technique can be viewed as a deadlock prevention scheme. The system does not know that there is a deadlock, but suspects there might be one and therefore aborts a transaction

Another deadlock prevention method is to run a test at the time the scheduler is about to block T_i because it is requesting a lock that conflicts with one owned by T_j. The test should guarantee that if the scheduler allows T_i to wait for T_j, then deadlock cannot result

Consists of eliminating one of the conditions that allows the possibility of deadlock when designing the concurrency control algorithm

50



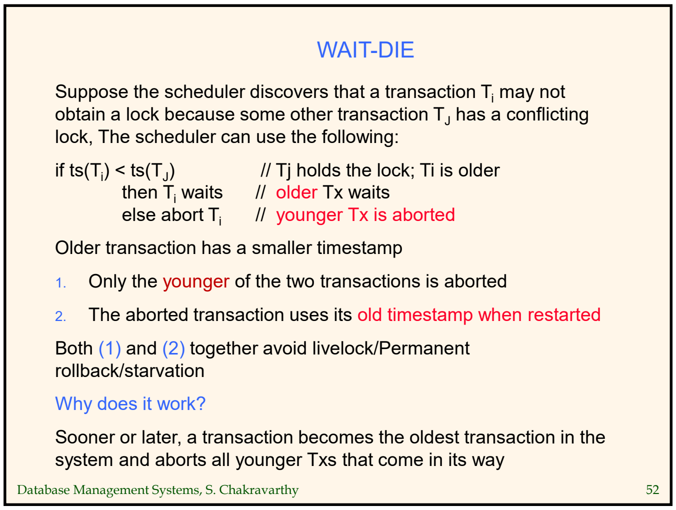
Wait-Die and Wound-Wait (deadlock prevention schemes)

The basic idea is to avoid the creation of a cycle in the wait-for-graph. Such avoidance is achieved, not by creating and inspection of the graph, but by introducing a suitable protocol that makes such cycles impossible!

These two are considered locking techniques, although time stamps are used for the purposes of choosing the victim. Assumptions:

- Every transaction gets a unique time-stamp, implies
  - No two transactions are started simultaneously
- Priority is the inverse of its time-stamp. Thus the older a transaction, the higher its priority

51



WAIT-DIE

Suppose the scheduler discovers that a transaction T_i may not obtain a lock because some other transaction T_J has a conflicting lock, The scheduler can use the following:

```
if ts(T_i) < ts(T_J)          // Tj holds the lock; Ti is older
         then T_i waits       //  older Tx waits
         else abort T_i       //  younger Tx is aborted
```

Older transaction has a smaller timestamp

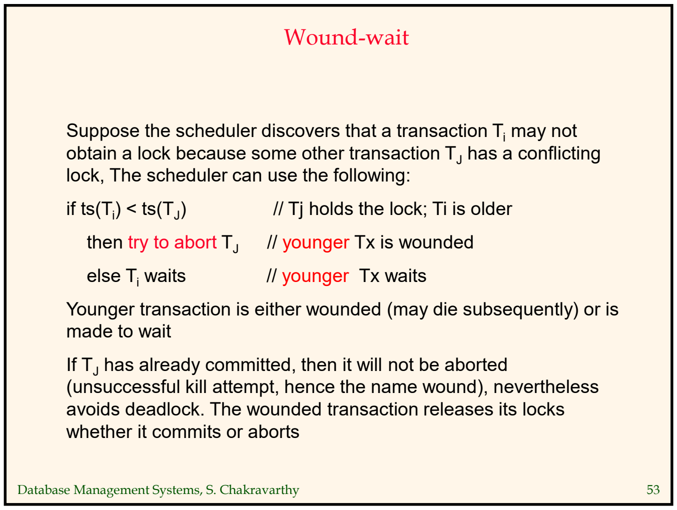
1. Only the younger of the two transactions is aborted
2. The aborted transaction uses its old timestamp when restarted

Both (1) and (2) together avoid livelock/Permanent rollback/starvation

Why does it work?

Sooner or later, a transaction becomes the oldest transaction in the system and aborts all younger Txs that come in its way

52



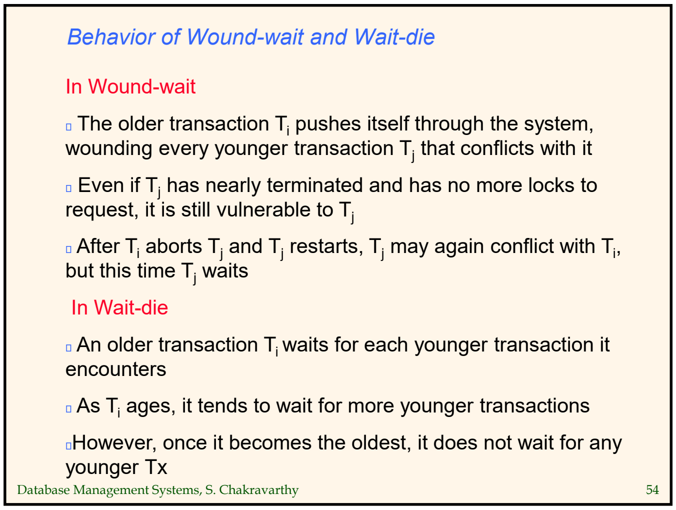
Wound-wait

Suppose the scheduler discovers that a transaction T_i may not obtain a lock because some other transaction T_J has a conflicting lock, The scheduler can use the following:

```
if ts(T_i) < ts(T_J)          // Tj holds the lock; Ti is older
   then try to abort T_J      // younger Tx is wounded
   else T_i waits             // younger  Tx waits
```

Younger transaction is either wounded (may die subsequently) or is made to wait

If T_J has already committed, then it will not be aborted (unsuccessful kill attempt, hence the name wound), nevertheless avoids deadlock. The wounded transaction releases its locks whether it commits or aborts



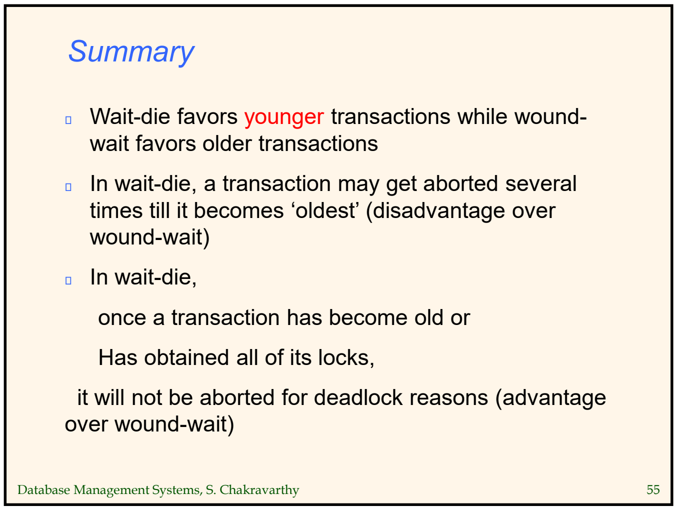
Behavior of Wound-wait and Wait-die

In Wound-wait

- The older transaction T_i pushes itself through the system, wounding every younger transaction T_j that conflicts with it
- Even if T_j has nearly terminated and has no more locks to request, it is still vulnerable to T_j
- After T_i aborts T_j and T_j restarts, T_j may again conflict with T_i, but this time T_j waits

In Wait-die

- An older transaction T_i waits for each younger transaction it encounters
- As T_i ages, it tends to wait for more younger transactions
- However, once it becomes the oldest, it does not wait for any younger Tx



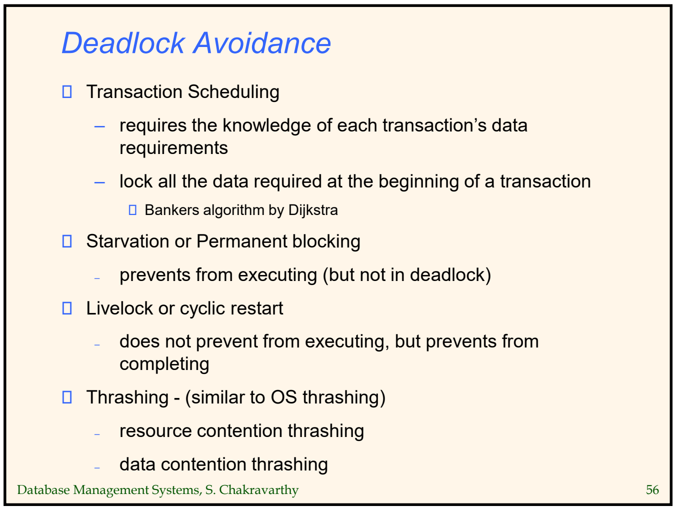
Summary

- Wait-die favors younger transactions while wound-wait favors older transactions
- In wait-die, a transaction may get aborted several times till it becomes ‘oldest’ (disadvantage over wound-wait)
- In wait-die,

  once a transaction has become old or

  Has obtained all of its locks,

it will not be aborted for deadlock reasons (advantage over wound-wait)



Deadlock Avoidance

- Transaction Scheduling
  - requires the knowledge of each transaction’s data requirements
  - lock all the data required at the beginning of a transaction
    - Bankers algorithm by Dijkstra
- Starvation or Permanent blocking
  - prevents from executing (but not in deadlock)
- Livelock or cyclic restart
  - does not prevent from executing, but prevents from completing
- Thrashing - (similar to OS thrashing)
  - resource contention thrashing
  - data contention thrashing

Conservative 2PL

It is possible to construct a 2PL scheduler that never aborts transaction. This technique is called conservative 2PL or static 2PL

Avoids deadlocks by requiring each transaction to obtain all of its locks before any operations are submitted to DM

This can be achieved by having each transaction predeclare its readset and writeset

Alternatively, transactions can be preanalyzed (conservatively, of course) to obtain its readset and writeset

The scheduler tries to set all of the locks needed by T_i. If all the locks cannot be obtained then T_i is made to wait

Every time the scheduler releases the locks of a completed transaction, it examines the waiting queue to see if it can grant all of the lock requests of any waiting transaction

- In conservative 2PL, transactions that are waiting hold no locks

Hence no deadlock and no aborts due to deadlock

Strict 2PL

- Almost all commercial implementations of 2PL use a variant called strict 2PL
- In strict 2PL, the scheduler releases all of a transaction's locks together, when the transaction terminates. Specifically, T_i's locks are released after DM acknowledges the processing of c_i or a_i
- To release lock(s) prior to the termination, the scheduler must know (to release ol_i [x]) :
  1. T_i has set all the locks it will ever need, and
  2. T_i will not subsequently issue operations that refer to x

Termination satisfies (1) and (2)

It is not easy to derive (1) and (2) before termination, in general

Actually,

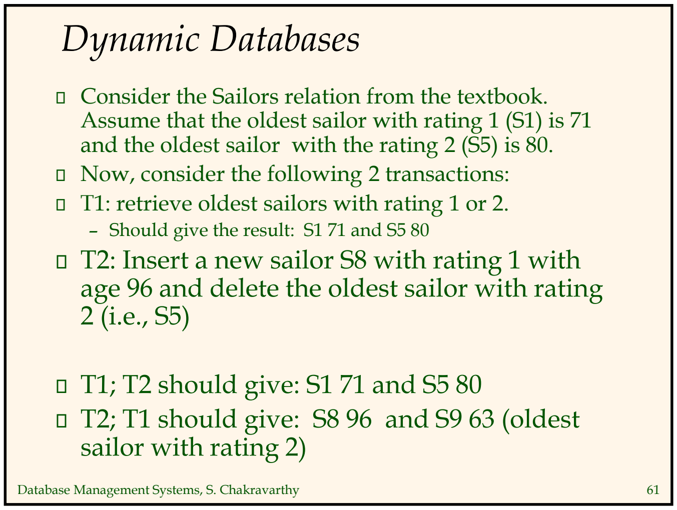
- It is only necessary to hold write locks until after a transaction commits or aborts to ensure strictness
- Read locks may be released earlier subject to the 2PL rules to ensure serializability
- Pragmatically, read locks can be released when the transaction terminates, but write locks must be held until after the transaction commits or aborts

Strict histories have nice properties

- They are recoverable
- They avoid cascading aborts
- Abort can be implemented by restoring before images

Miscellany

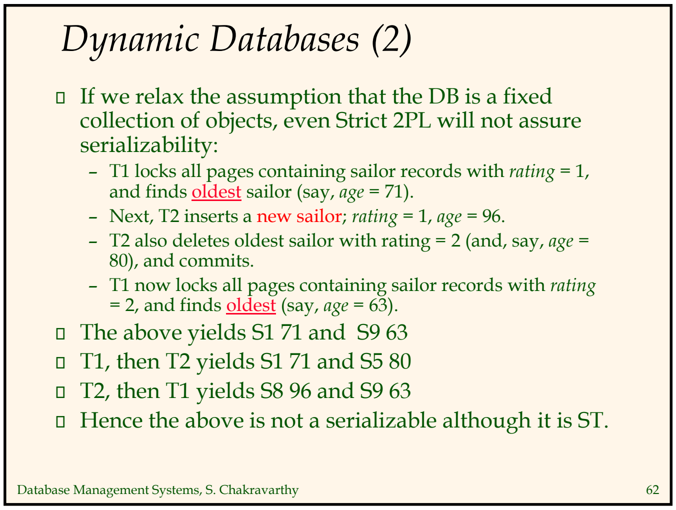
- The Phantom Problem
  - Concurrency control problem for dynamic databases
- The convoy phenomenon
  - A preemptive scheduler preempting a process requiring a high traffic resource can create a convoy (Interaction between OS scheduler and high traffic resource)
- Hot spot
  - heavy write traffic data items e.g. lock manager, total of branch accounts, log, etc.
- Halloween problem please look it up as an exercise



Dynamic Databases

- Consider the Sailors relation from the textbook. Assume that the oldest sailor with rating 1 (S1) is 71 and the oldest sailor with the rating 2 (S5) is 80.
- Now, consider the following 2 transactions:
- T1: retrieve oldest sailors with rating 1 or 2.
  - Should give the result: S1 71 and S5 80
- T2: Insert a new sailor S8 with rating 1 with age 96 and delete the oldest sailor with rating 2 (i.e., S5)
- T1; T2 should give: S1 71 and S5 80
- T2; T1 should give: S8 96 and S9 63 (oldest sailor with rating 2)

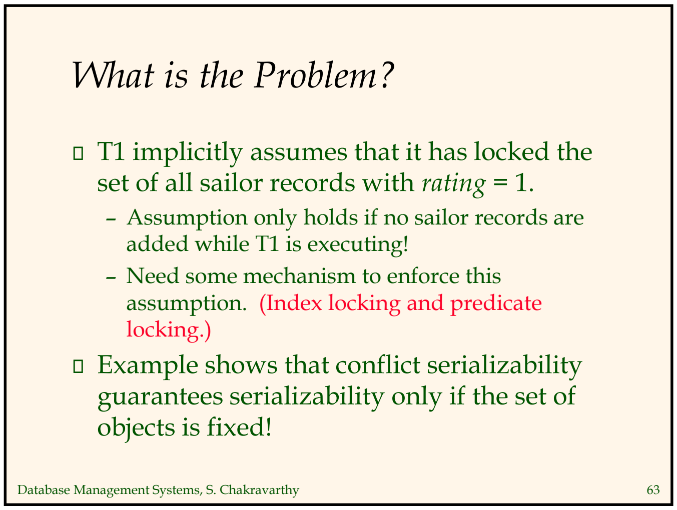
61



Dynamic Databases (2)

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
  - T2 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- The above yields S1 71 and S9 63
- T1, then T2 yields S1 71 and S5 80
- T2, then T1 yields S8 96 and S9 63
- Hence the above is not a serializable although it is ST.

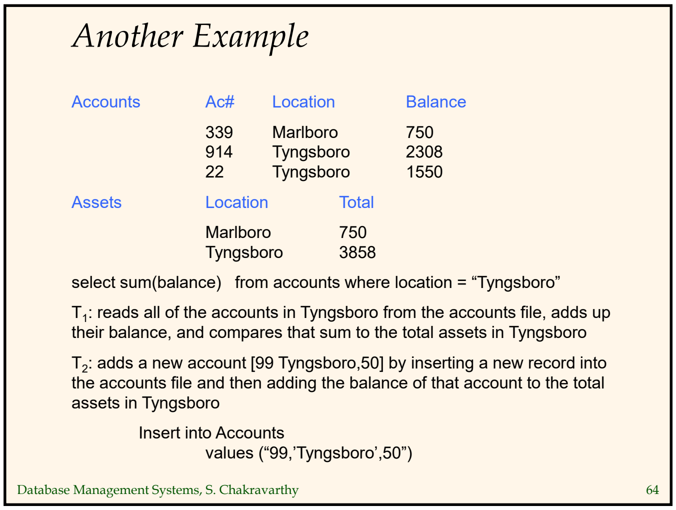
62



What is the Problem?

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

63



Another Example

Accounts	Ac#	Location	Balance
	339	Marlboro	750
	914	Tyngsboro	2308
	22	Tyngsboro	1550

Assets	Location	Total
	Marlboro	750
	Tyngsboro	3858

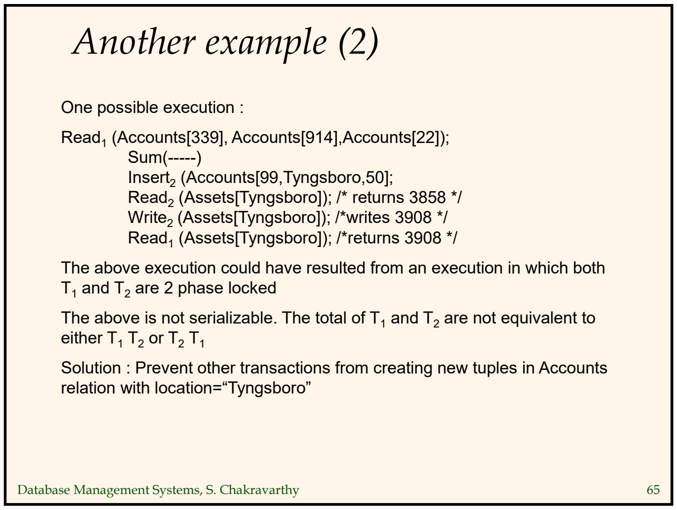
select sum(balance) from accounts where location = “Tyngsboro”

T_1: reads all of the accounts in Tyngsboro from the accounts file, adds up their balance, and compares that sum to the total assets in Tyngsboro

T_2: adds a new account [99 Tyngsboro,50] by inserting a new record into the accounts file and then adding the balance of that account to the total assets in Tyngsboro

```
Insert into Accounts
        values (“99,’Tyngsboro’,50”)
```

64



Another example (2)

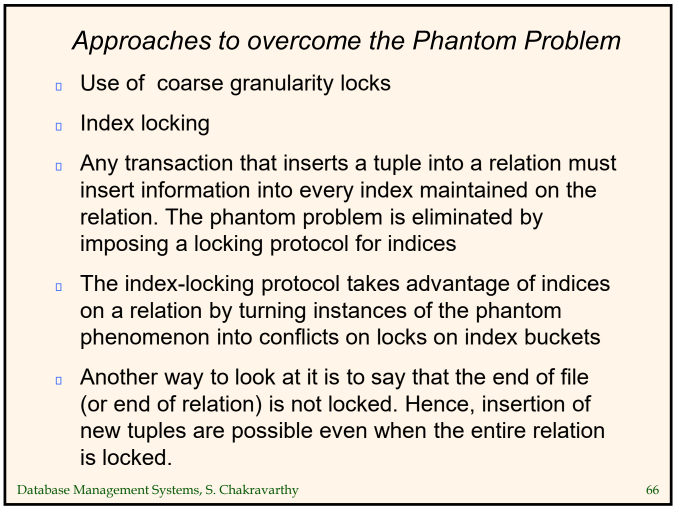
One possible execution :

Read$_1$ (Accounts[339], Accounts[914],Accounts[22]);
Sum(-----)
Insert$_2$ (Accounts[99,Tyngsboro,50];
Read$_2$ (Assets[Tyngsboro]); /* returns 3858 */
Write$_2$ (Assets[Tyngsboro]); /*writes 3908 */
Read$_1$ (Assets[Tyngsboro]); /*returns 3908 */

The above execution could have resulted from an execution in which both T$_1$ and T$_2$ are 2 phase locked

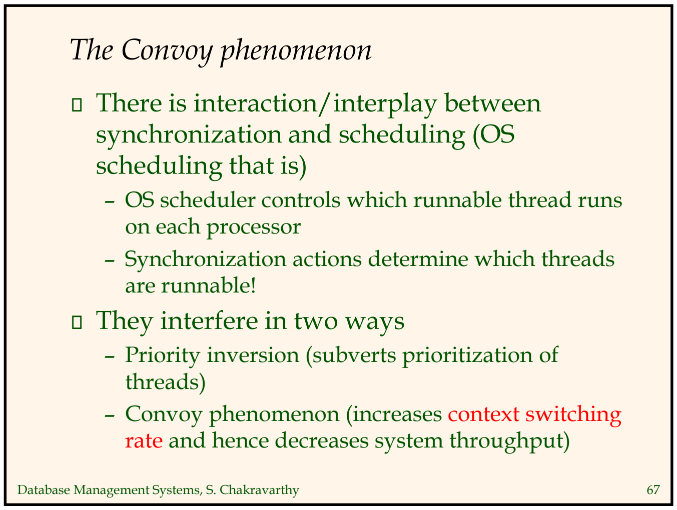
The above is not serializable. The total of T$_1$ and T$_2$ are not equivalent to either T$_1$ T$_2$ or T$_2$ T$_1$

Solution : Prevent other transactions from creating new tuples in Accounts relation with location="Tyngsboro"



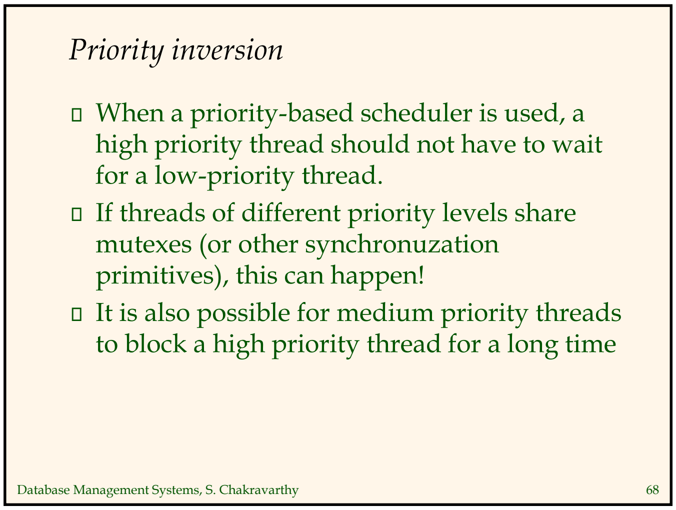
Approaches to overcome the Phantom Problem

- Use of coarse granularity locks
- Index locking
- Any transaction that inserts a tuple into a relation must insert information into every index maintained on the relation. The phantom problem is eliminated by imposing a locking protocol for indices
- The index-locking protocol takes advantage of indices on a relation by turning instances of the phantom phenomenon into conflicts on locks on index buckets
- Another way to look at it is to say that the end of file (or end of relation) is not locked. Hence, insertion of new tuples are possible even when the entire relation is locked.



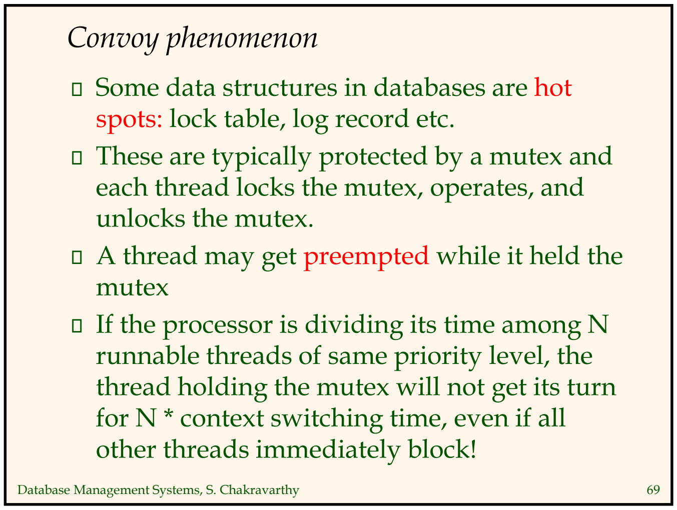
The Convoy phenomenon

- There is interaction/interplay between synchronization and scheduling (OS scheduling that is)
  - OS scheduler controls which runnable thread runs on each processor
  - Synchronization actions determine which threads are runnable!
- They interfere in two ways
  - Priority inversion (subverts prioritization of threads)
  - Convoy phenomenon (increases context switching rate and hence decreases system throughput)



Priority inversion

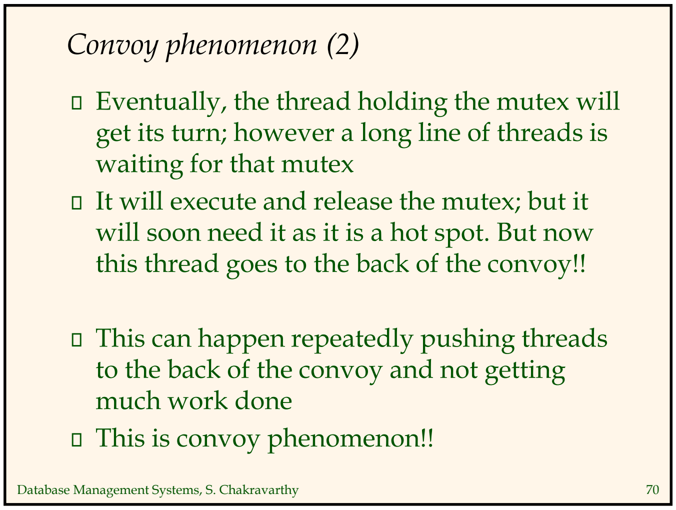
- When a priority-based scheduler is used, a high priority thread should not have to wait for a low-priority thread.
- If threads of different priority levels share mutexes (or other synchronuzation primitives), this can happen!
- It is also possible for medium priority threads to block a high priority thread for a long time



Convoy phenomenon

- Some data structures in databases are hot spots: lock table, log record etc.
- These are typically protected by a mutex and each thread locks the mutex, operates, and unlocks the mutex.
- A thread may get preempted while it held the mutex
- If the processor is dividing its time among N runnable threads of same priority level, the thread holding the mutex will not get its turn for N * context switching time, even if all other threads immediately block!

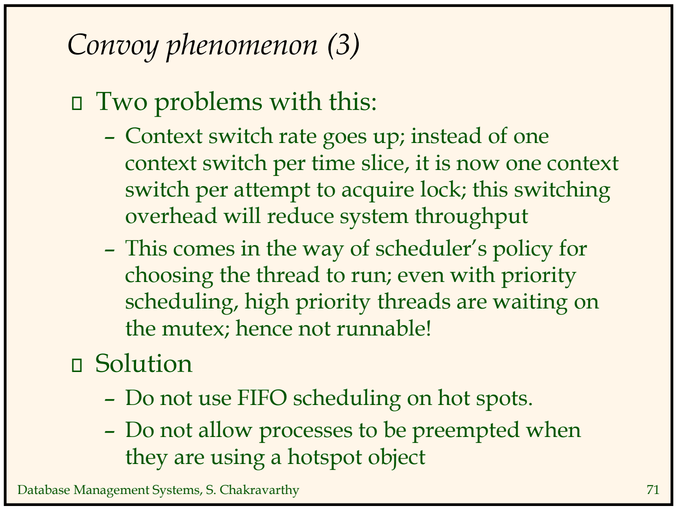
69



Convoy phenomenon (2)

- Eventually, the thread holding the mutex will get its turn; however a long line of threads is waiting for that mutex
- It will execute and release the mutex; but it will soon need it as it is a hot spot. But now this thread goes to the back of the convoy!!
- This can happen repeatedly pushing threads to the back of the convoy and not getting much work done
- This is convoy phenomenon!!

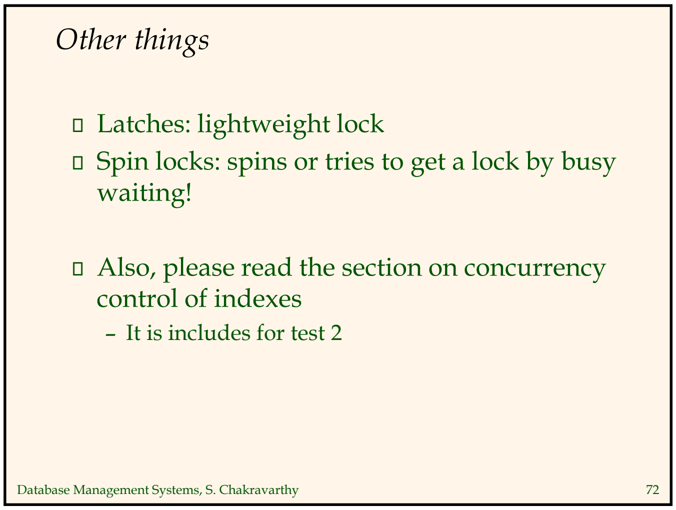
70



Convoy phenomenon (3)

- Two problems with this:
  - Context switch rate goes up; instead of one context switch per time slice, it is now one context switch per attempt to acquire lock; this switching overhead will reduce system throughput
  - This comes in the way of scheduler’s policy for choosing the thread to run; even with priority scheduling, high priority threads are waiting on the mutex; hence not runnable!
- Solution
  - Do not use FIFO scheduling on hot spots.
  - Do not allow processes to be preempted when they are using a hotspot object

71



Other things

- Latches: lightweight lock
- Spin locks: spins or tries to get a lock by busy waiting!
- Also, please read the section on concurrency control of indexes
  - It is includes for test 2

72

Thank You !