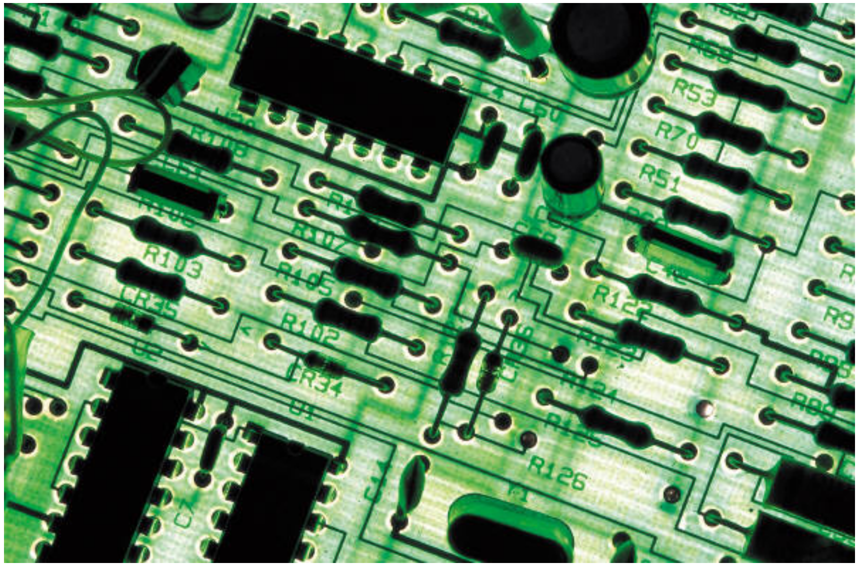
COSC344
Database Theory and Applications



Lecture 22
Concurrency Control

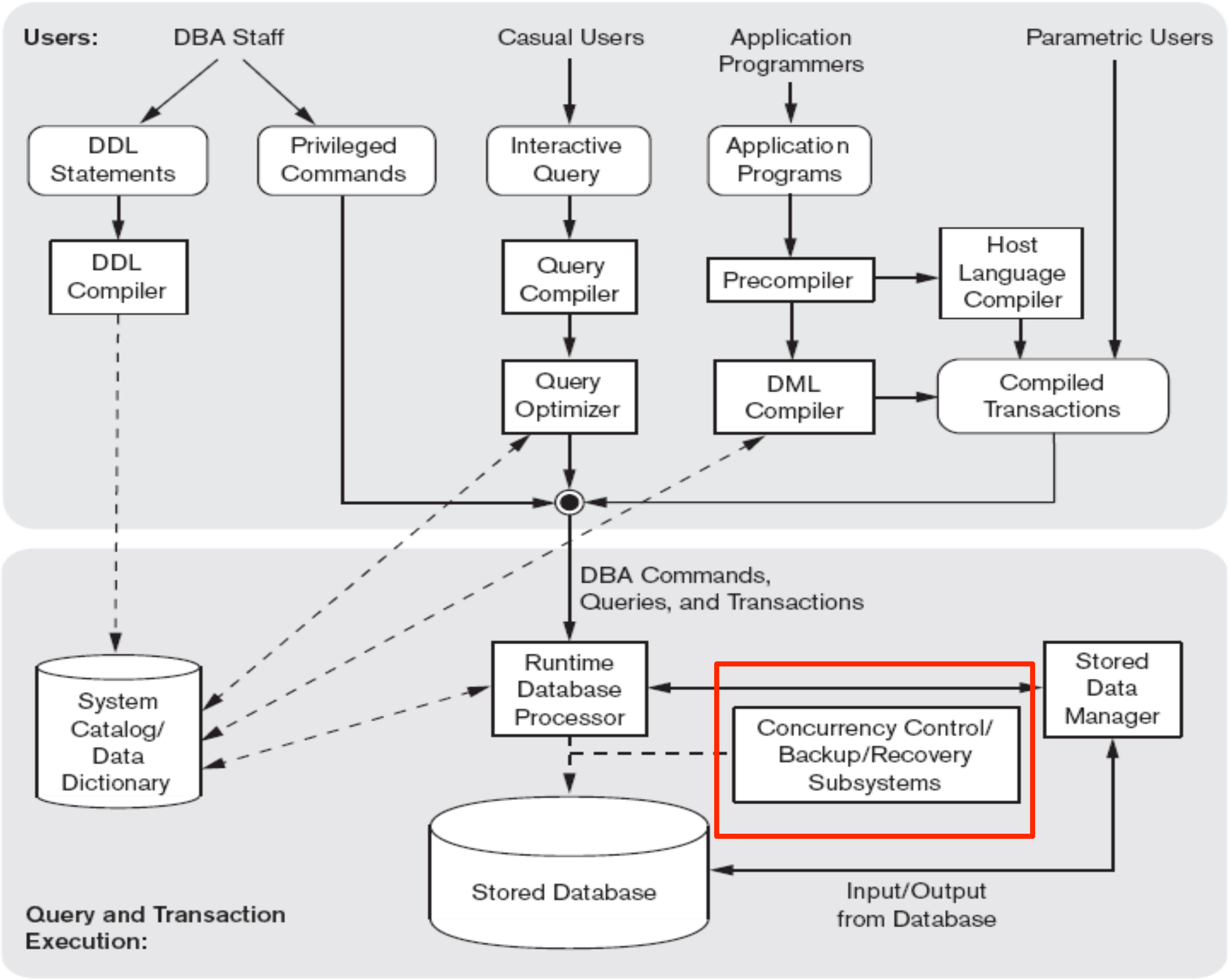
Overview

- Last Lecture
  - Transactions
- This Lecture
  - Concurrency control
  - **Source: Chapter 21**
- Next Lecture
  - Database recovery
  - Source: Chapter 22

Question to Ponder

- Effects of allowing concurrent access to a database? a booking system with many simultaneous users

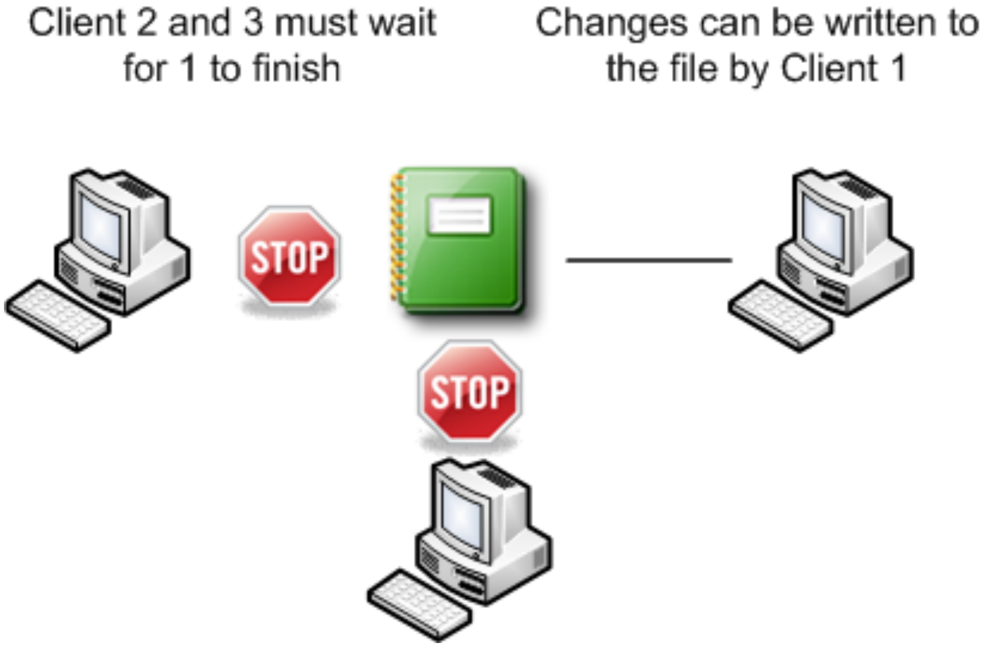
DBMS Component Modules
Where are we now?



Lecture 17

Figure 2.3
Component modules of a DBMS and their interactions.

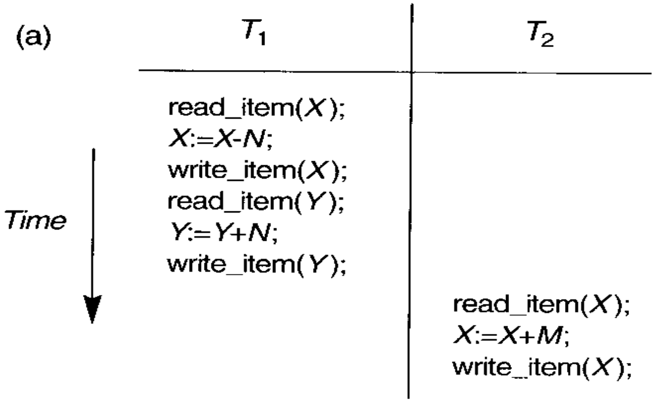
Review - Concurrency Control

- Concurrency: Interleaving of __transactions__________
- Three Concurrency problems
  - _Lost update_____ problem
  - _Dirty read______ problem
  - Incorrect summary problem
- 

Review - Serialisability

- Serialisable: equivalent to some <u>serial schedule</u>

(a)



T_1	T_2

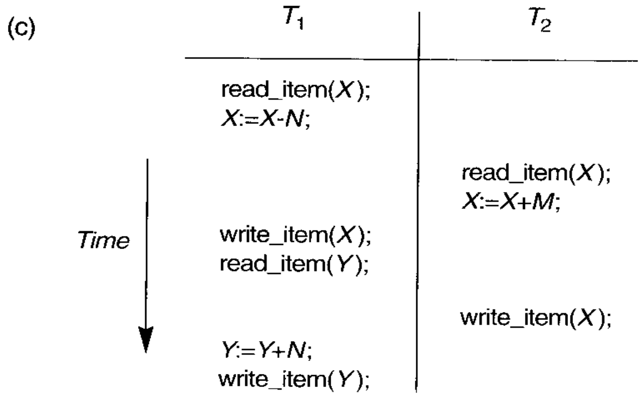
read_item(X);
X:=X-N;
write_item(X);
read_item(Y);
Y:=Y+N;
write_item(Y);	

	read_item(X);
	X:=X+M;
	write_item(X);

Schedule *A*

✓

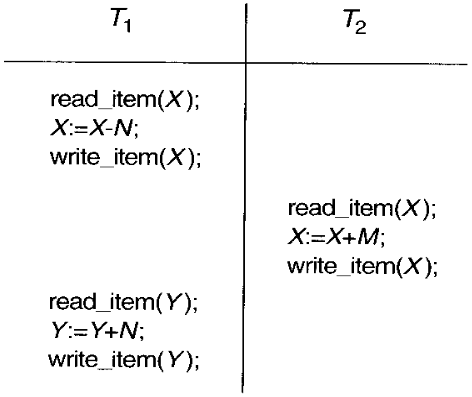
(c)



T_1	T_2
read_item(X); X:=X-N;	
	read_item(X); X:=X+M;
write_item(X); read_item(Y);	
	write_item(X);
Y:=Y+N; write_item(Y);	

Schedule *C*

✗



T_1	T_2

read_item(X);
X:=X-N;
write_item(X);	

	read_item(X);
	X:=X+M;
	write_item(X);

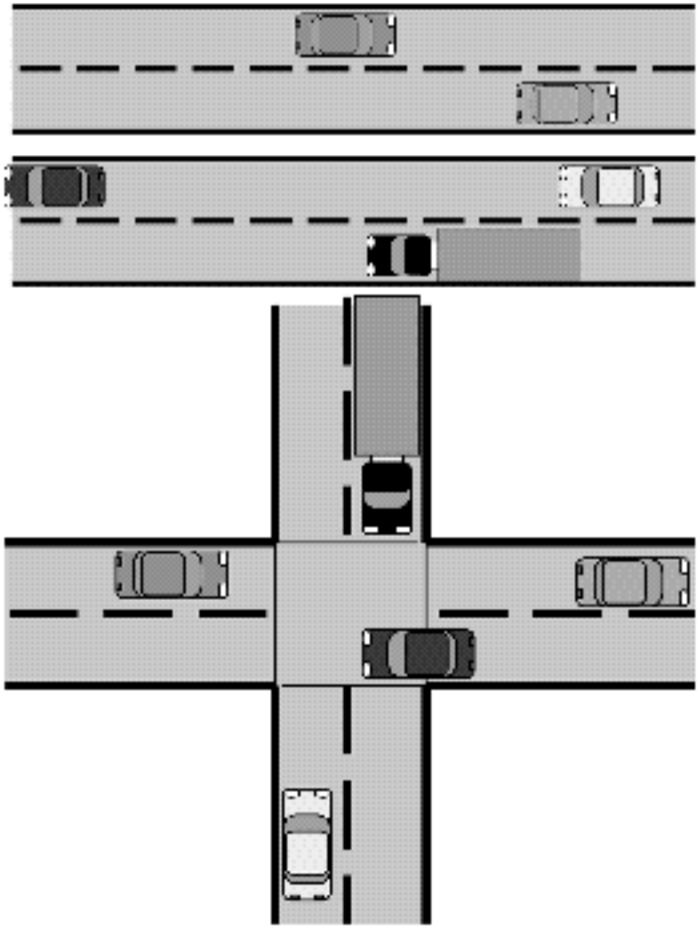
read_item(Y);
Y:=Y+N;
write_item(Y);	

Schedule *D*

✓

Concurrency Control

Develop protocols which ensure serialisability – Locking



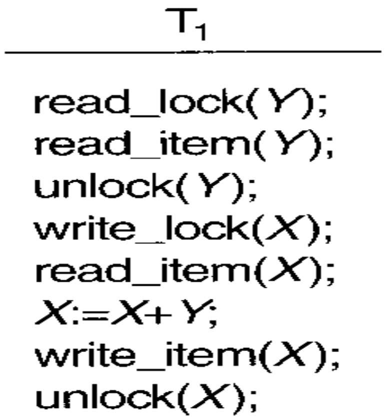
Locking

- Basic Idea: acquires a _lock___ on that object
- Two kinds of locks
  - Write (W) (exclusive)
  - Read (R) (shared)
- Locking Rules
  - issue a read_lock(x) or a write_lock(x) before read_item(x).
  - issue a write_lock(x) before write_item(x).
  - issue an unlock(x) after all read_item(x) and write_item(x) are completed.

```
T1
read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);
```

Locking protocols are used in most Commercial DBMSs

Locking Compatibility Matrix



```
T_1
read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);
```

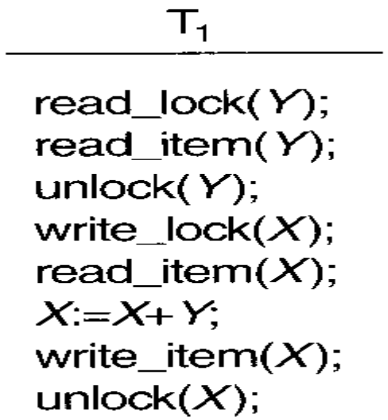
Transaction A holds this kind of lock

Transaction B requests this kind of lock

	W	R
W	N	N
R	N	Y

Locking Algorithm

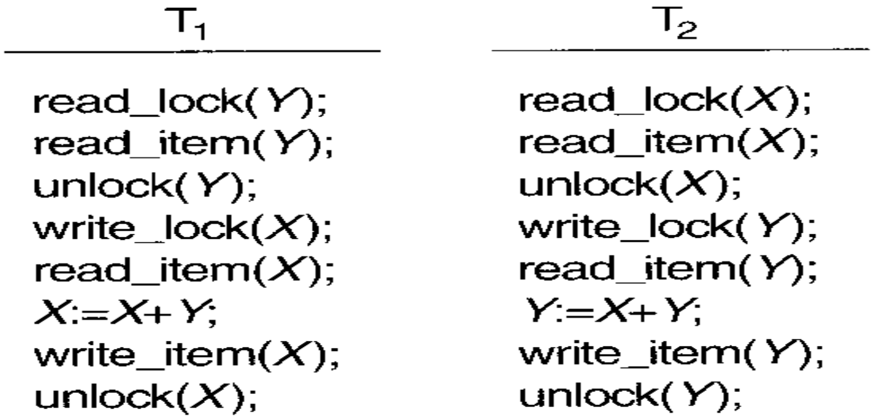
- If transaction A holds a write lock on *p*, then a request from other transactions for a lock of either type on *p* will be denied.
- If transaction A holds a read lock on *p*, then:
  - request from other transactions for a write lock on *p* will be____denied_____.
  - request from other transactions for a read lock on *p* will be ____granted_____.



```
T1
read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);
```

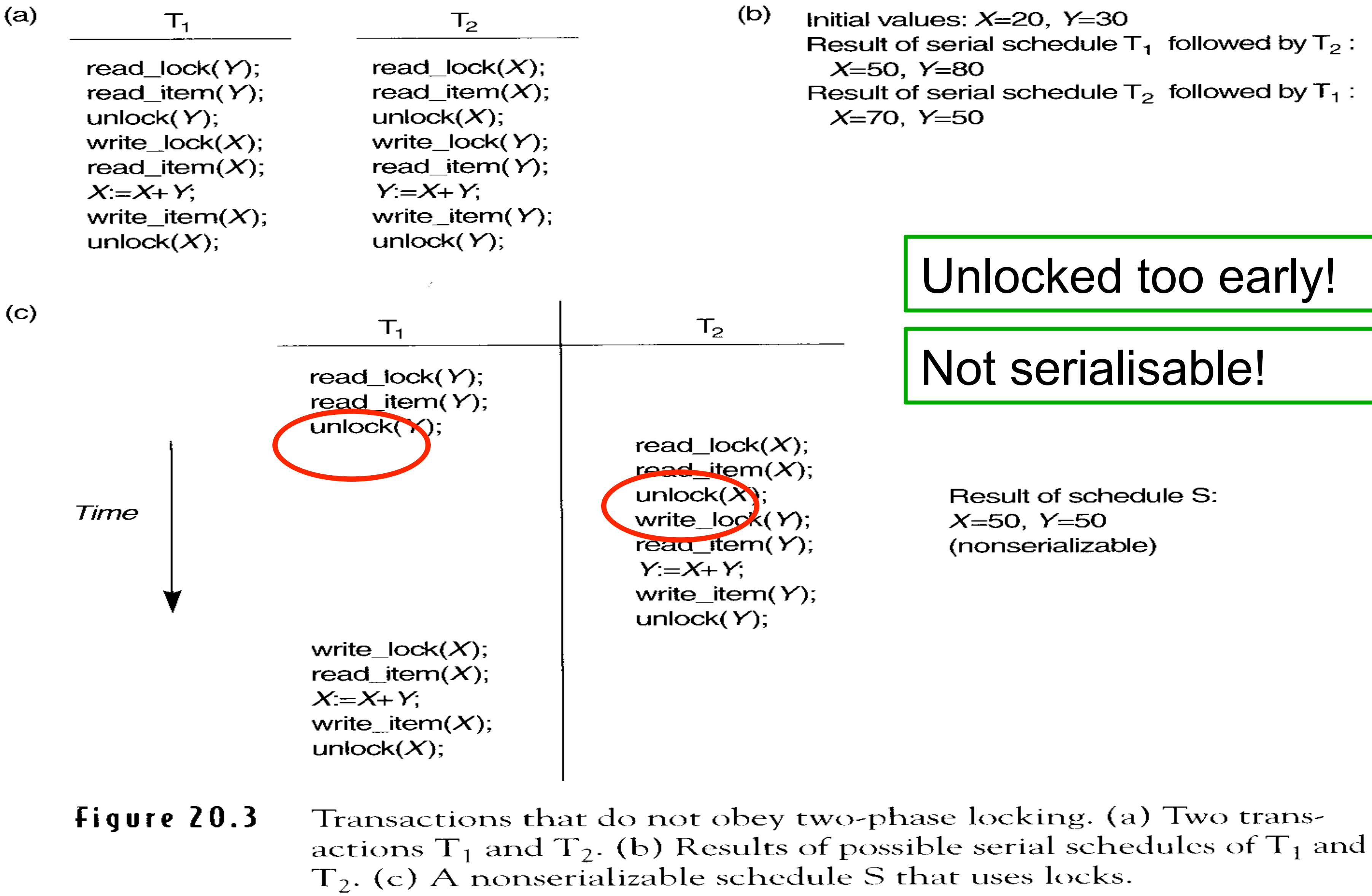
Data Access Protocol with Locking

- to read p, first acquire a __read__ lock on p.
- to update p, first acquire a __write__ lock on p.
- If a lock request by B is denied because it conflicts with a lock already held by A, B goes into a __wait__ state. B will wait until A's lock is __released__.
  - The system must guarantee that B does not wait forever
    - can use a queue for lock requests



T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X:=X+Y$;	$Y:=X+Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Example



(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X:=X+Y$;	$Y:=X+Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b) Initial values: $X=20$, $Y=30$

Result of serial schedule T_1 followed by T_2 :

$X=50$, $Y=80$

Result of serial schedule T_2 followed by T_1 :

$X=70$, $Y=50$

(c)

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y:=X+Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X:=X+Y$;	
write_item(X);	
unlock(X);	

Result of schedule S:

$X=50$, $Y=50$

(nonserializable)

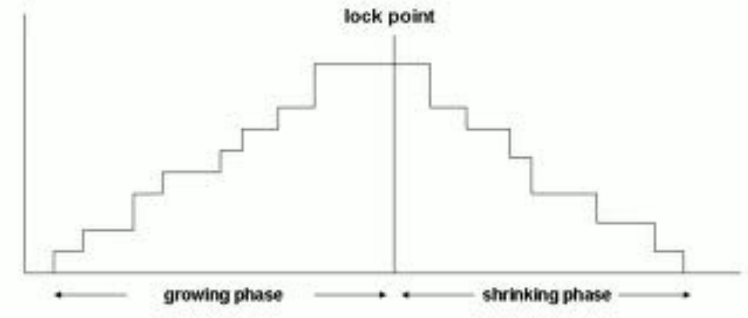
Figure 20.3 Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

Unlocked too early!

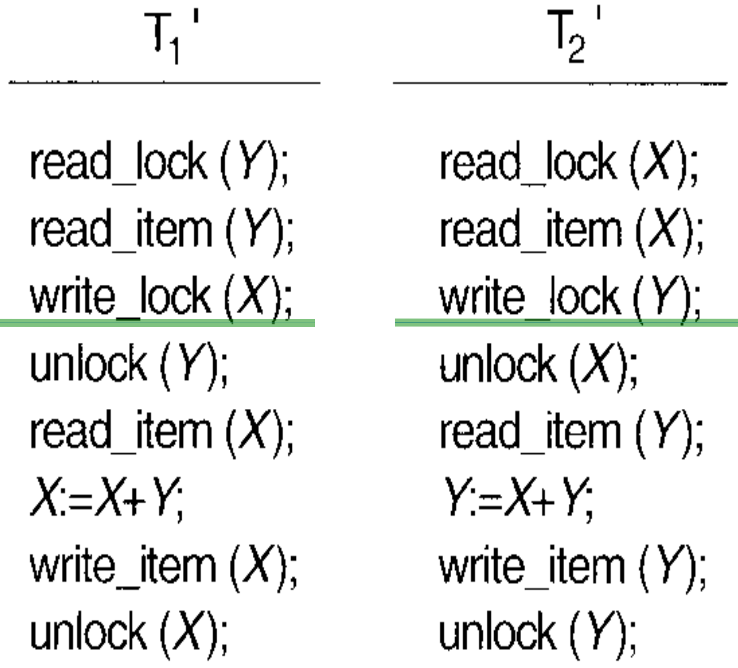
Not serialisable!

Two-Phase Locking

- Protocol
  - Before operating on any object, must acquire a lock on that object.
  - After releasing a lock, must never go on to acquire any more locks.
- Two phases
  - Expanding - acquiring locks
  - Shrinking - releasing locks
- Theorem
  - If all transactions obey the ______two-phase locking______ protocol, then all possible interleaved schedules are serialisable



Two-Phase Locking Example



Expanding phase

T_1'	T_2'
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	write_lock (Y);

Shrinking phase

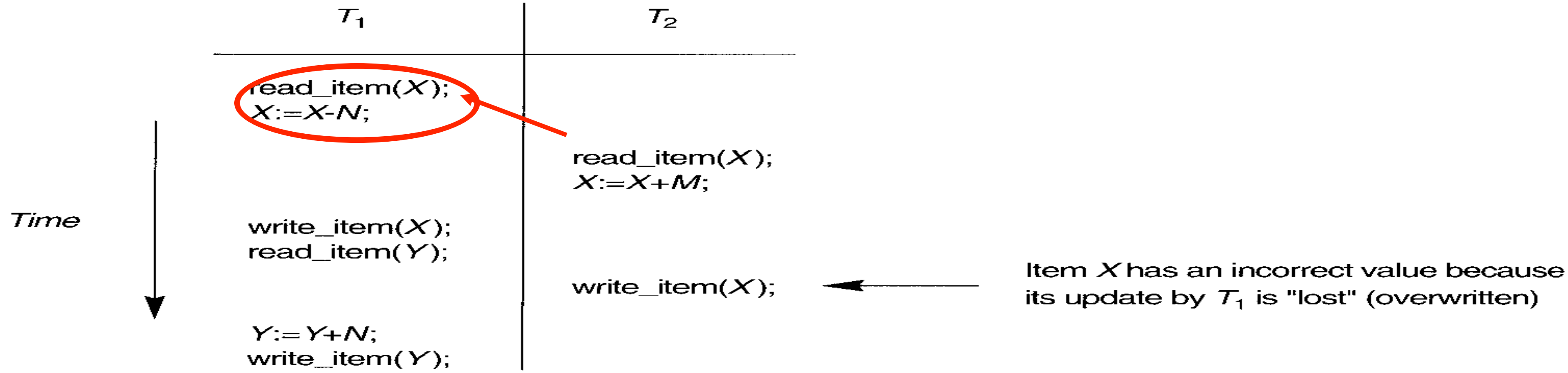
unlock (Y);	unlock (X);
read_item (X);	read_item (Y);
$X:=X+Y$;	$Y:=X+Y$;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

Figure 20.4 Transactions T_1' and T_2', which are the same as T_1 and T_2 of Figure 20.3 but which follow the two-phase locking protocol. Note that they can produce a deadlock.

Variations of Two-Phase Locking

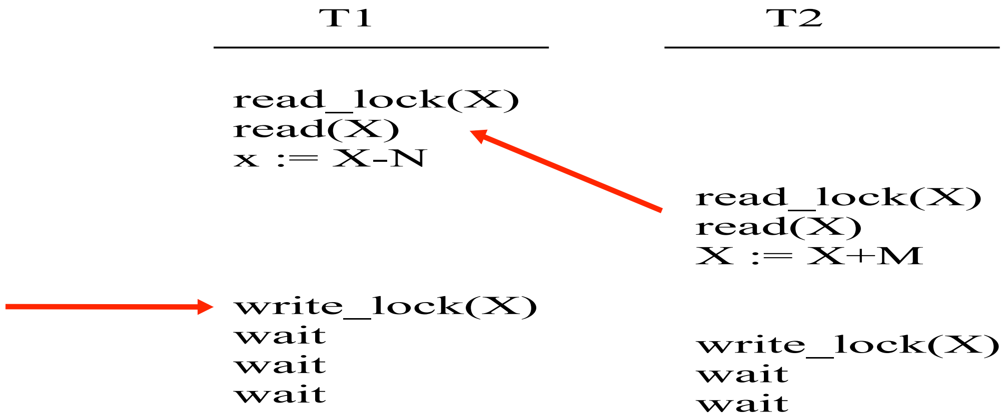
- Conservative 2PL
  - Locks all items before the transaction begins
- Strict 2PL   (most common)
  - Does not release any Write locks until commit or rollback.

Lost Update Problem Revisited



T_1	T_2
read_item(X); X:=X-N;	
	read_item(X); X:=X+M;
write_item(X); read_item(Y);	
	write_item(X);
Y:=Y+N; write_item(Y);	

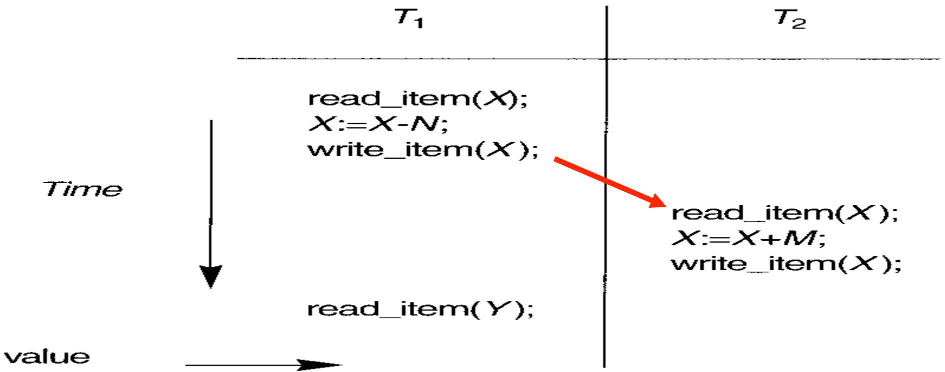
Item X has an incorrect value because its update by T_1 is "lost" (overwritten)



T1	T2
read_lock(X) read(X) x := X-N	
	read_lock(X) read(X) X := X+M
write_lock(X) wait wait wait	
	write_lock(X) wait wait

A new problem - deadlock!

Temporary Update or Uncommitted Dependency Problem Revisited



	T_1	T_2

Time	read_item(X);
	X:=X-N;
	write_item(X);	

		read_item(X);
		X:=X+M;
		write_item(X);

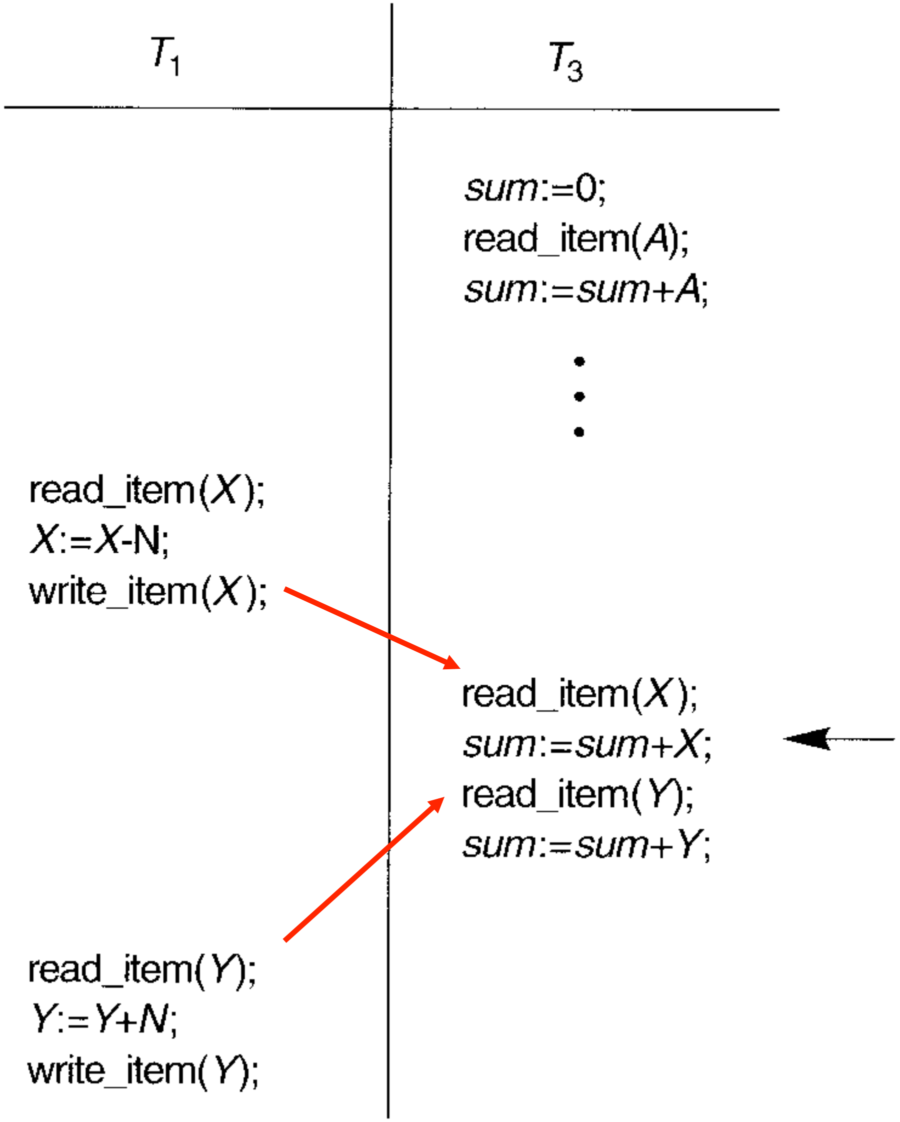
	read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the "temporary" incorrect value of X.

Transaction fails - rollback

T1	T2
read_lock(X)	
read(X)	
X := X-N	
write_lock(X)	
write(X)	
	read_lock(X)
	wait
	wait
read_lock(Y)	wait
read(Y)	wait
rollback	wait
	read(X)

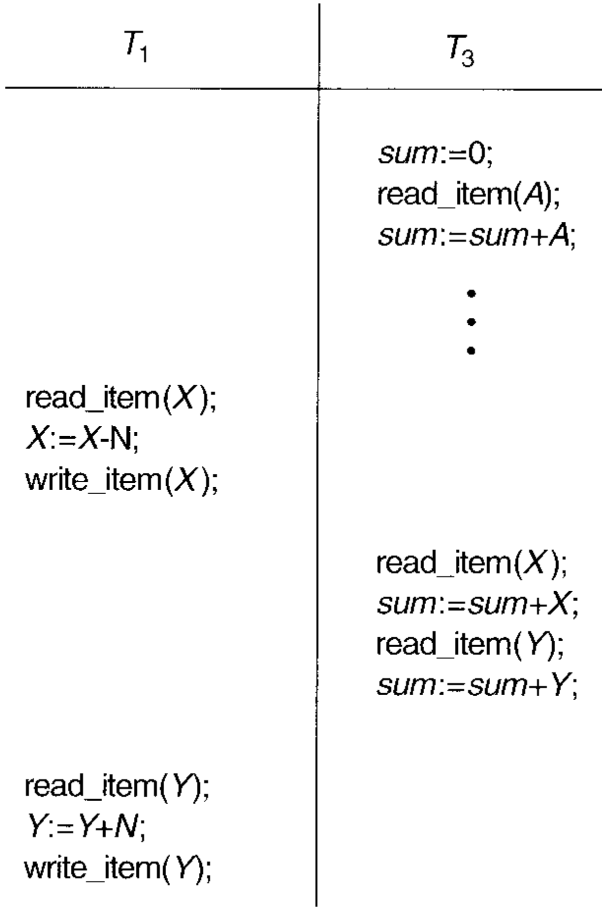
Incorrect Summary Problem Revisited



T_1	T_3
	sum:=0;
	read_item(A);
	sum:=sum+A;
	⋮
read_item(X);	
X:=X-N;	
write_item(X);	
	read_item(X);
	sum:=sum+X;
	read_item(Y);
	sum:=sum+Y;
read_item(Y);	
Y:=Y+N;	
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Incorrect Summary Problem Revisited (continued)



T_1	T_3

	sum:=0;
	read_item(*A*);
	sum:=*sum*+*A*;
	⋮

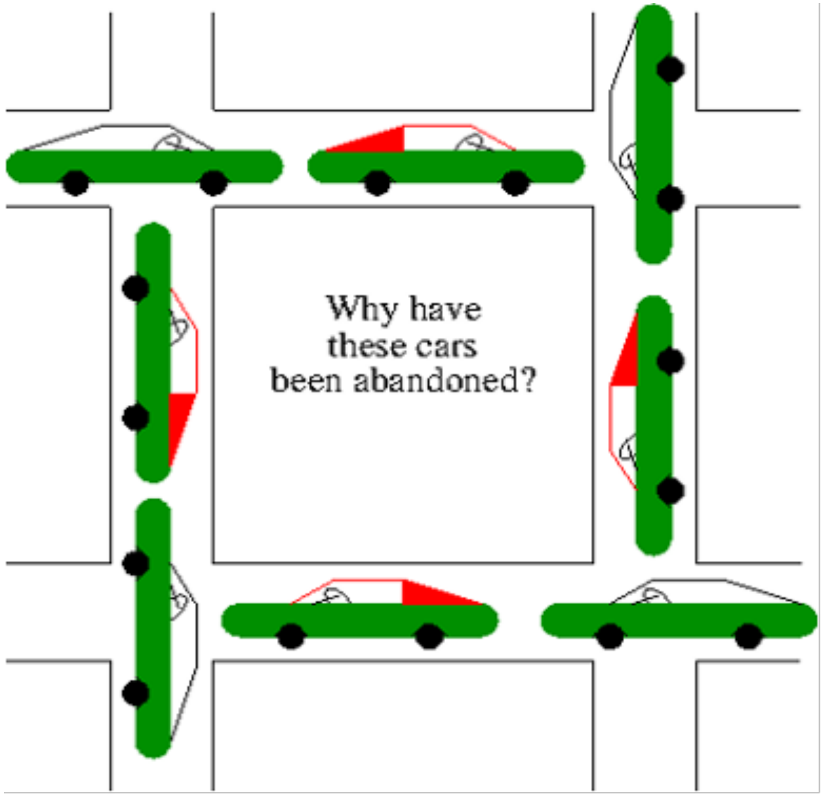
read_item(*X*);
X:=*X*-N;
write_item(*X*);	

	read_item(*X*);
	sum:=*sum*+*X*;
	read_item(*Y*);
	sum:=*sum*+*Y*;

read_item(*Y*);
Y:=*Y*+*N*;
write_item(*Y*);	

T1	T3
	sum := 0
	read_lock(A)
	read(A)
	sum := sum+A
	. . .
	. . .
read_lock(X)	
read(X)	
X := X-N	
write_lock(X)	
write(X)	
	read_lock(X)
	wait
	wait
read_lock(Y)	wait
read(Y)	wait
Y := Y+N	wait
write_lock(Y)	wait
write(Y)	wait
commit (and	wait
release locks)	wait
	read(X)

Deadlock

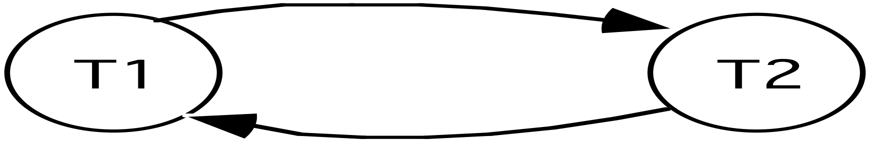


Deadlock

- What is it?
- Two approaches
  - Prevention
  - Detection

T1	T2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
	write_lock(Y)

time

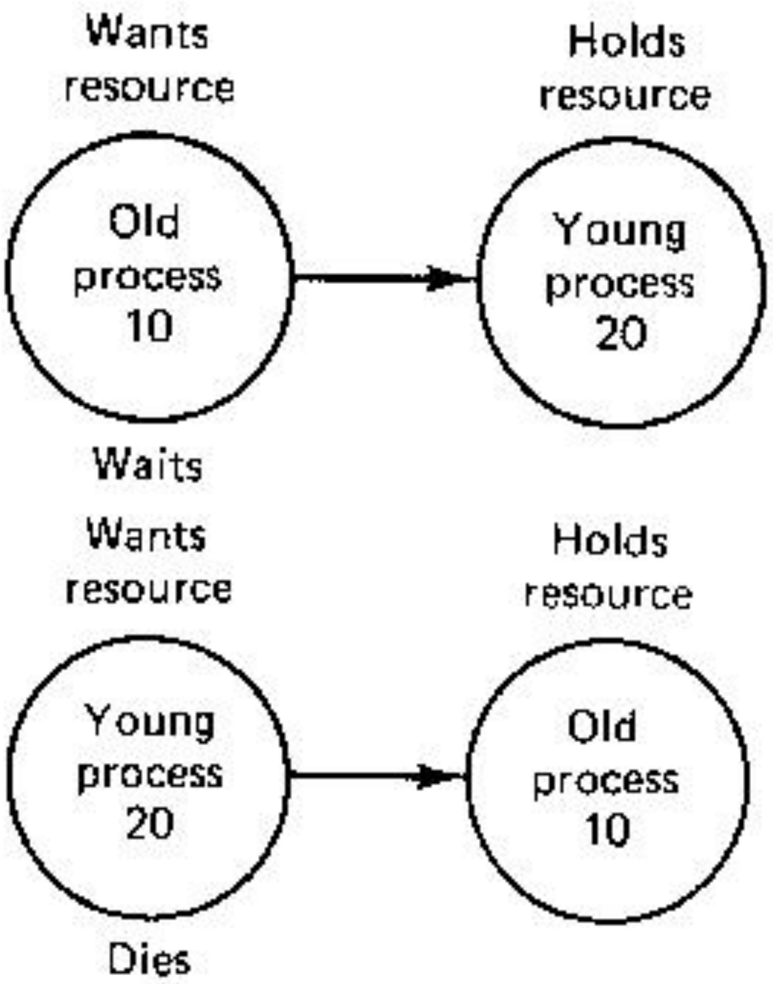


A wait-for graph

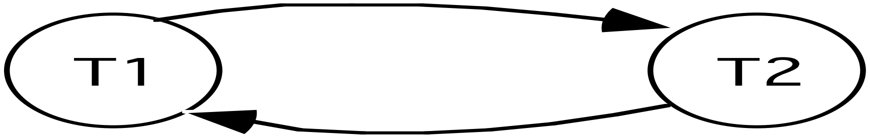
Timestamp-Based Deadlock Prevention

- Transaction timestamp TS(T)
  - ordered based on the start of a transaction (give priority)
- $TS(T_1) < TS(T_2)$ if T1 started first
  - T1 is the older transaction

- Wait-Die - *older transaction waits on younger*
  - If $TS(T_1) < TS(T_2)$, T_1 waits
    - *(T_1 is older than T_2)*
  - Otherwise, abort T_1 (T_1 dies) and restart it with the same timestamp
    - *(T_1 is younger than T_2)*



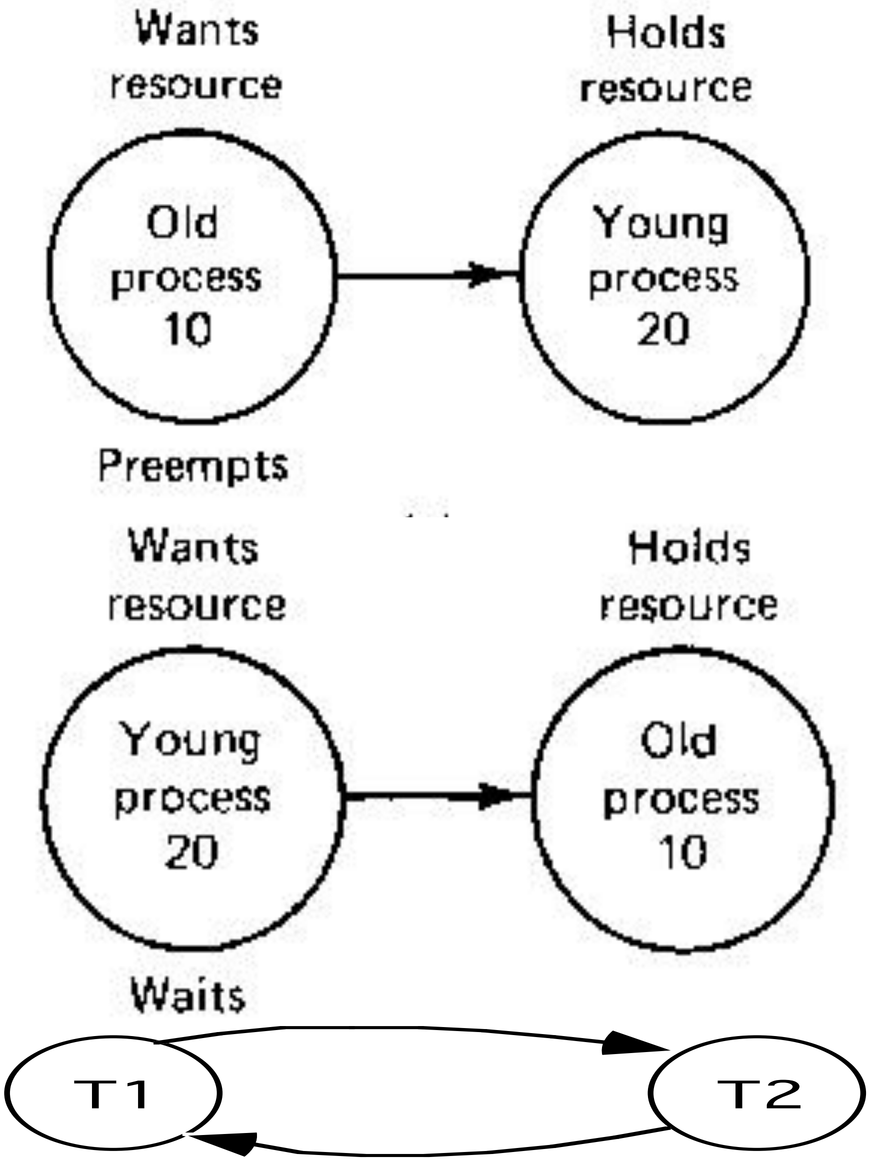
Wait-Die kills the younger transaction!



Timestamp-Based Deadlock Prevention

- Transaction timestamp TS(T)
  - ordered based on the start of a transaction
- $TS(T_1) < TS(T_2)$ if T1 started first
  - T1 is the older transaction

- Wound-Wait - *younger transaction waits on older*
  - If $TS(T_1) < TS(T_2)$, abort T_2 (T_1 wounds T_2) and restart it later with the same timestamp
    - *(T_1 is older than T_2)*
  - Otherwise, T_1 waits
    - *(T_1 is younger than T_2)*



Wound-Wait preempts the younger transaction!

Non Time-Based Deadlock Prevention Schemes

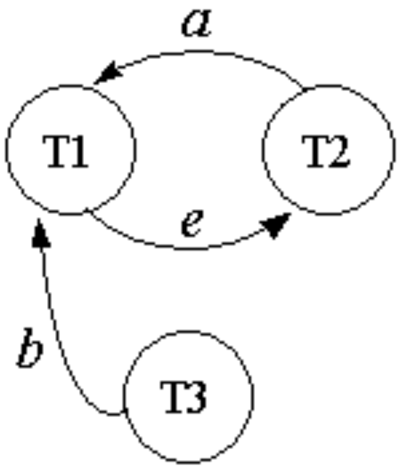
© Original Artist / Search ID: rmon2093

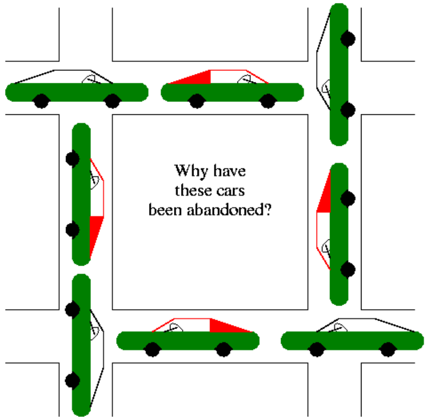
Rights Available from CartoonStock.com

- No waiting
  - – If a transaction cannot acquire a lock, it is aborted and restarted after a time delay without checking whether a deadlock will actually occur. *lots of needless aborts and restarts*
- Cautious waiting
  - – Assume T_1 tries to acquire a lock on X but X is locked by T_2.
  - – If T_2 is not blocked, then T_1 is blocked and allowed to wait; otherwise abort T_1.
- Timeouts
  - – If a transaction waits longer than a set time, deadlock is assumed and it is aborted.

Deadlock Detection

- Construct a wait-for graph
  - A node for each __transaction______
  - A directed edge (T1 -> T2) whenever _T1__ is waiting to lock an item that is locked by ___T2___
  - A deadlock is indicated by a __cycle___.





suits short transactions, little interference between transactions, accessing a few items