COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 20 EXERCISES
CONCURRENCY CONTROL: LOCK-BASED PROTOCOLS

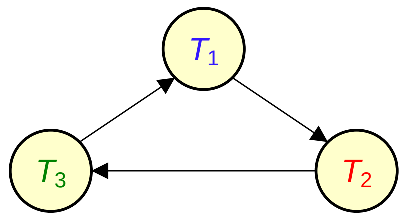
EXERCISE 1

a) Is the schedule conflict serializable?

If yes, give the equivalent serial schedule.

T_1	T_2	T_3
read(X)		
	read(Y)	
	write(Y)	
		write(Z)
write(X)		
	read(X)	
	write(X)	
		read(Y)
		write(Y)
write(Z)		

Precedence graph



The schedule is <u>not conflict serializable</u> because there is a cycle T_1 T_2 T_3 T_1 in the precedence graph.

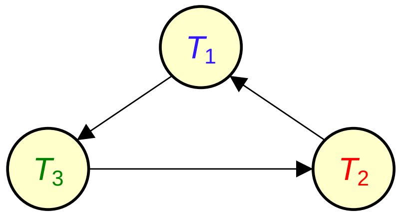
Therefore, the schedule will fail under any protocol that aims at conflict serializability.

EXERCISE 1 (CONT'D)

T_1	T_2	T_3
lock-s(X) ✓		
read(X)		
	lock-s(Y) ✓	
	read(Y)	
	lock-x(Y) ✓	
	write(Y)	
		lock-x(Z) ✓
		write(Z)
lock-x(X) ✓		
write(X)		
	lock-s(X) ✗ cannot be granted – T_2 must wait	
	read(X)	
	write(X)	
		lock-s(Y) ✗ cannot be granted – T_3 must wait
		read(Y)
		write(Y)
lock-x(Z) ✗ cannot be granted – T_1 must wait		
write(Z)	**deadlock!**	

b) Modify the schedule according to <u>strict 2PL</u> by adding a lock-s() before reading a data item, a lock-x() before writing a data item and an unlock() after all read/write operations have completed. Is the schedule deadlock free?

Wait-for graph



<u>Strict 2PL</u>

All exclusive-mode locks must be held until a transaction commits (shared-mode locks can be released anytime).

Cannot request any lock after releasing a lock.

EXERCISE 2

Which of the following statements is true about the wait-for graph (circle the correct answer)?

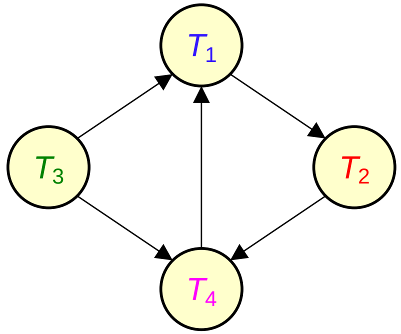
a) T_4 is waiting for T_3 to release a data item.

b) The system is in a deadlock state after removing the edge between T_2 and T_4.

c) The system is in a deadlock state after removing the edge between T_3 and T_4.

d) The system is in a deadlock state when T_1 no longer holds a data item needed by T_4.

Wait-for graph



a) is not correct since T_3 is waiting for T_4.

b) and d) are not correct since after removing these edges, there does not exist a cycle in the wait-for graph.

EXERCISE 3

Show that the following schedule is conflict serializable according to 2PL by adding lock-s(), lock-x() and unlock() instructions, as necessary, to the schedule. If possible, add the lock-s(), lock-x() and unlock() instructions so that *no transaction is required to wait*.

T_1	T_2	T_3
read(X)		
	read(X)	
		read(Y)
read(Z)		
	read(Y)	
	write(X)	
		read(X)
		write(X)
write(Z)		

EXERCISE 3 (CONT'D)

T_1	T_2	T_3
lock-s(X) read(X) unlock(X)		
	lock-s(X) read(X)	
		lock-s(Y) read(Y)
lock-x(Z) read(Z)		
	lock-s(Y) read(Y)	
	lock-x(X) write(X) unlock(Y) unlock(X)	
		lock-s(X) read(X)
		lock-x(X) write(X) unlock(Y) unlock(X)
write(Z) unlock(Z)		

Do you see any problem with this schedule?

Under 2PL, a transaction cannot request locks after it has released any lock.

EXERCISE 3 (CONT'D)

T_1	T_2	T_3
lock-s(X) read(X)		
	lock-s(X) read(X)	
		lock-s(Y) read(Y)
lock-x(Z) read(Z)		
	lock-s(Y) read(Y)	
	lock-x(X) write(X) unlock(Y) unlock(X)	
		lock-s(X) read(X)
		lock-x(X) write(X) unlock(Y) unlock(X)
write(Z) unlock(Z) unlock(X)		

Do you see any problem with this schedule?

This lock cannot be granted; T_2 must wait.

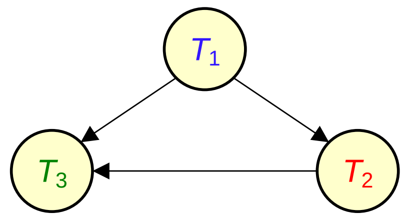
How to add the locks so that T_2 does not have to wait?

T_1	T_2	T_3
lock-s(X) ✓ read(X)		
	lock-s(X) ✓ read(X)	
		lock-s(Y) ✓ read(Y)
lock-x(Z) ✓ unlock(X) ✓ read(Z)		
	lock-s(Y) ✓ read(Y)	
	lock-x(X) ✓ write(X) unlock(Y) ✓ unlock(X) ✓	
		lock-s(X) ✓ read(X)
		lock-x(X) √ write(X) unlock(Y) ✓ unlock(X) ✓
write(Z) unlock(Z) ✓		

EXERCISE 3 (cont'd)

Recall: Under 2PL, a transaction cannot request locks <u>after</u> it has released any lock.

Precedence graph



The schedule is <u>conflict serializable</u>.

The equivalent serial schedule is T_1 T_2 T_3.

EXERCISE 4

In which positions, A to E, can an unlock(X) instruction be inserted if the schedule is according to:

a) strict 2PL (circle the correct answer)

⟹ all x-locks must be held until a transaction commits

⟹ X has only s-locks in the example

i. {A} {B} {C} {D}

ii. {A} {B} {C} {D} {E}

iii. {A} {C} {D}

iv. {B} {E}

v. {A} {C} {D} {E}

T_1	T_2
lock-s(X)	
read(X)	
	lock-s(X)
lock-x(Y)	
{A}	
read(Y)	
write(Y)	
	read(X)
	{C}
commit	
unlock(Y)	
{B}	
	{D}
	commit
	{E}

EXERCISE 4 (CONT'D)

In which positions, A to E, can an unlock(X) instruction be inserted if the schedule is according to:

b) rigorous 2PL (circle the correct answer)
⟹ all locks must be held until a transaction commits

i. {A} {B} {C} {D}
ii. {A} {B} {C} {D} {E}
iii. {A} {C} {D}
iv. {B} {E}
v. {A} {C} {D} {E}

T_1	T_2
lock-s(X)	
read(X)	
	lock-s(X)
lock-x(Y)	
{A}	
read(Y)	
write(Y)	
	read(X)
	{C}
commit	
unlock(Y)	
{B}	
	{D}
	commit
	{E}

EXERCISE 5

a) Is the schedule conflict serializable? If yes, give the equivalent serial schedule.

b) If T_3 aborts after write(Y), which other transactions will be rolled back?

c) If T_1 aborts after write(X), which other transactions will be rolled back?

d) Construct the wait-for graph that results from this schedule if all locks are only exclusive-locks (lock-x), no locks are released, and the execution process runs to the point of lock-x(Y) in T_1.

e) Add lock-s(), lock-x() and unlock() instructions to the schedule according to strict 2PL. Indicate which transactions, if any, are required to wait and which instructions cause the wait.

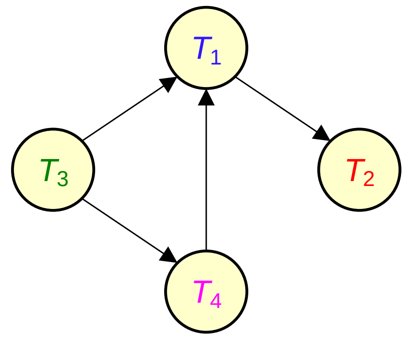
T_1	T_2	T_3	T_4
read(X)			
write(X)			
	read(X)		
		read(Y)	
		write(Y)	
	write(X)		
			read(Y)
write(Y)			

EXERCISE 5 (cont'd)

a) Is the following schedule conflict serializable? If yes, give the equivalent serial schedule.

T_1	T_2	T_3	T_4
read(X)			
write(X)			
	read(X)		
		read(Y)	
		write(Y)	
	write(X)		
			read(Y)
write(Y)			

Precedence graph



The schedule is conflict serializable. The equivalent serial schedule is T_3 T_4 T_1 T_2.

EXERCISE 5 (cont'd)

b) If T_3 aborts at the end of this schedule, which other transactions will be rolled back?

T_4 because after write(Y) in T_3, the Y read by T_4 is corrupted. Note that the write(Y) of T_1 is not affected by T_3 as it is a blind write (i.e., no read before write) and in the serialization order the write(Y) of T_1 would come after the write(Y) of T_3 and overwrite it.

c) If T_1 aborts at the end of this schedule, which other transactions will be rolled back?

T_2 because after write(X) in T_1, the X read by T_2 is corrupted.

T_1	T_2	T_3	T_4
read(X)			
write(X)			
	read(X)		
		read(Y)	
		write(Y)	
	write(X)		
			read(Y)
write(Y)			

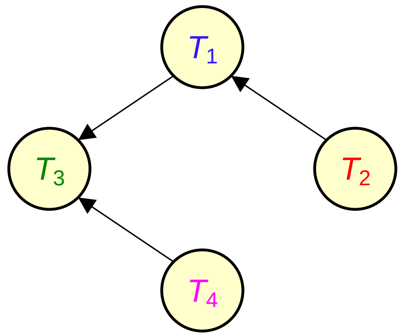
The equivalent serial schedule is T_3 T_4 T_1 T_2.

EXERCISE 5 (cont'd)

d) Construct the wait-for graph that results from this schedule if all locks are only exclusive-locks (lock-x), no locks are released, and the execution process runs to the point of lock-x(Y) in T_1.

T_1	T_2	T_3	T_4
lock-x(X) ✓			
read(X)			
write(X)			
	lock-x(X) ✗ cannot be granted – T_2 must wait		
	read(X)		
		lock-x(Y) ✓	
		read(Y)	
		write(Y)	
	write(X)		
			lock-x(Y) ✗ cannot be granted – T_3 must wait
			read(Y)
lock-x(Y) ✗ cannot be granted – T_1 must wait			
write(Y)			

Wait-for Graph



EXERCISE 5 (cont'd)

e) Add lock-s(), lock-x() and unlock() instructions to the schedule according to strict 2PL. Indicate which transactions, if any, are required to wait and which instructions cause the wait.

Recall: Under strict 2PL, a transaction must hold all its cvajsqg c-k mbc locks until it commits.

Normally, a transaction will request a lock-s before it reads a data item and upgrade to a lock-x later if it writes the data item.

T_1	T_2	T_3	T_4
lock-s(X)			
read(X)			
lock-x(X)			
write(X)			
	lock-s(X)		
		lock-s(Y)	
		read(Y)	
		lock-x(Y)	
		write(Y)	
		commit	
		unlock(Y)	
			lock-s(Y)
	T_2 waits		read(Y)
			commit
			unlock(Y)
lock-x(Y)			
write(Y)			
commit			
unlock(Y)			
unlock(X)			
	read(X)		
	lock-x(X)		
	write(X)		
	commit		
	unlock(X)		