Homework 4
Concurrency Control

Problem 1 *Consider the following transaction $T1$:*

$$R(A);W(A);R(B);W(B).$$

1. Give an example of another transaction ($T2$) that, if run concurrently to transaction $T1$ without some form of concurrency controlm could interfere with $T1$.
2. Explain, how the use of Strict 2PL would prevent interference between the transactions.

Problem 2 *Consider a database with objects A and B and assume that there are two transactions $T1$ and $T2$. Transaction $T1$ reads objects A and B and then writes object B. Transaction $T2$ reads objects A and B and then writes objects A and B.*

1. Give an example schedule with actions of transactions $T1$ and $T2$ on objects A and B that results in a write-read conflict.
2. Give an example schedule with actions of transactions $T1$ and $T2$ on objects A and B that results in a read-write conflict.
3. Give an example schedule with actions of transactions $T1$ and $T2$ on objects A and B that results in a write-write conflict.
4. Show that Strict 2PL disallows each of the three schedules above.

Problem 3 *All schedules below must contain at least 3 transactions.*

1. Give an example of a serializable schedule that would not be accepted by Strict 2PL mechanism.

2. Give an example of a conflict-serializable schedule that would not be accepted by Strict 2PL mechanism.

3. Give an example of a strict schedule that would not be accepted by Strict 2PL mechanism.

Problem 4 *For each of the schedules below, specify whether it is serial, serializable, conflict-serializable, view-serializable, strict. Qualify your answer; construct dependency (serializability) graphs for each schedule.*

6.1

T1	T2	T3	T4
R(A)			
	R(A)		
		W(B)	
R(B)			
			R(C)
	commit		
W(C)			
		W(C)	
			W(B)
W(A)			
		R(A)	
commit			
			W(C)
		commit	
			commit

6.2

T1	T2
R(A)	
R(B)	
W(B)	
W(C)	
W(C)	
commit	
	W(B)
	R(C)
	R(D)
	W(A)
commit	

6.3

T1	T2	T3
R(A)		
W(B)		
	R(B)	
		R(B)
	W(A)	
		R(A)
R(A)		
	abort	
W(A)		
		W(A)
commit		commit

6.4

T1	T2	T3
R(A)		
	W(B)	
R(B)		
	R(A)	
W(B)		
commit		
		W(B)
		W(A)
		commit
	commit	

6.5

T1	T2	T3
R(C)		
		R(A)
R(B)		
W(A)		
	R(A)	
commit		
	R(B)	
		R(C)
	W(C)	
	commit	
		R(C)
		W(C)
		W(A)
		commit

6.6

T1	T2	T3
R(A)		
	R(B)	
		R(C)
W(B)		
		W(C)
	W(A)	
	commit	
W(C)		
commit		
		W(A)
		commit

Problem 5 *For each of the* locking mechanisms *(*protocols*) below determine whether they ensure the following properties of schedules: serializability, conflict-serializability, recoverability. For each question, give a proof in case of a positive answer or a counterexample - in case of negative answer.*

- **Write Only Locking (WOL):** *Each transaction must obtain an exclusive lock before any write. All exclusive locks are released when transaction terminates (commits or aborts).* **No** *lock is needed to read an object.*

- **Liberal Read Locking (LRL):** *Each transaction must obtain an exclusive lock to write an object and a shared lock to read an object. Exclusive locks are preserved until the end of transaction, shared locks can be released and acquired at will.*

- **Onetime Object Locking (OOL):** *Each transaction n must obtain an exclusive lock to write an object and a shared lock to read an object. Locks of any kind are released at will, but the transaction cannot* ***re-obtain*** *a lock on the object on which it has already held (and then released) the lock.*

- **2 Phase Write Only Locking (2P WOL):** *Each transaction must obtain an exclusive lock before any write. A lock can be released at will, but after the first lock is released no new locks can be acquired.* **No** *lock is needed to read an object.*

- **Conservative Write/ Liberal Read Locking (CW/LRL):** *All exclusive locks must be requested by a transaction at the start (transaction either acquires all the exclusive locks it requested or is suspended and acquires no locks). All exclusive locks are released when transaction terminates. Shared locks can be acquired and released at will.*

Problem 6 *Show how transaction manager will execute the schedule below. At each step, show the contents of both* transaction *and* lock tables *and associated lists and queues. Assume* Strict 2 Phase Locking.

T1	*T2*	*T3*
X(B)		
	S(A)	
	R(A)	
		S(B)
S(A)		
R(A)		
	X(A)	
W(B)		
commit		
		R(B)
		X(A)
	W(A)	
	commit	
		W(A)
		commit

Problem 7 *Consider each of the schedules below. For each schedule, the deadlock resolution mechanism is indicated. For each schedule :*

- *Construct the* real schedule *which shows how the transactions will be executed under the deadlock resolution mechanism.*
- *Indicate every place where deadlock resolution mechanism will affects the execution of the schedule.*
- *Produce a succession of* ***waits-for*** *graphs for schedules managed by* ***deadlock detection*** *protocols. Aborted transactions are restarted immediately.*[1]

The order of transactions is always: $T1 > T2 > T3 > \ldots$ (i.e., $T1$ is always ***the oldest****).*

1. Protocol: **Strict 2 Phase Locking; Deadlock Prevention; Wound-Wait.**

   Schedule Request: T1:S(A); T2:S(A); T1:R(A); T3: X(B); T1:X(A); T3:S(A); T2:X(B); T1:commit; T3: R(A); T3:X(A); T3:W(A); T3:commit; T2:W(B); T2:commit.

2. Protocol: **Strict 2 Phase Locking; Deadlock Prevention; Wait-Die.**

[1] More than possible final schedule exists. Any "reasonable" correct schedule will be accepted.

Schedule Request: T1:S(A); T2:X(B); T1:R(A); T3: X(B); T1:X(B); T2:W(B); T2:S(A); T2:R(A); T2:X(A); T3:W(B); T3:commit; T1:W(B); T1:commit; T2:W(A); T2:commit.

3. Protocol: **Strict 2 Phase Locking; Deadlock Detection (after every third step); abort youngest transaction.**

   Schedule Request: T1:S(A); T2:S(A); T3:X(B); T1:R(A); T2:R(A); T3:R(B); T2:X(B); T3:W(B); T1:X(A); T3:W(B); T3:X(A); T3:W(A); T2:R(B); T2:W(B); T2:X(A); T1:W(A); T1:commit; T3:commit; T2: commit.

Note for Problem 7. Schedule Request reflects the schedule of actions as it originally was supposed to happen. When some transaction is aborted, the real schedule becomes different from the original schedule request. At this point, there is a number of possible ways to reschedule the actions from aborted (and then restarted) transactions. One possible way is to give each transaction a *turn* in which it will try to execute its current pending action (like it is done in **Problem 4**).