Two-phase Locking Protocol

2PL protocol ensures a serializable schedule. Let us look at the following:

T1	T2
read_lock(A)	
read(A)	
read_lock(B)	
read(B)	
	read_lock(A)
	read(A)
	unlock(A)
unlock(A)	
unlock(B)	

read_locks for both A and B were acquired.
No conflict operations.
The above schedule is serializable. (T1 -> T2)

Now consider the conflict operations in following non-serial schedule:

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
	read(B)
	write(B)
read(B)	
write(B)	
commit	
	commit

Using **Basic 2-phase locking protocol**, we get the following:

T1	T2
write_lock(A)	
read(A)	
write(A)	
	~~write_lock(A)~~
write_lock(B)	
read(B)	
write(B)	
unlock(A)	
unlock(B)	
	write_lock(A)
	read(A)
	write(A)
	write_lock(B)
	read(B)
	write(B)
	unlock(A)
	unlock(B)
commit	
	commit

Basic 2PL produces a serializable schedule.

Conflict: T2 read the value A written by T1.

What happens if T1 rollbacks rather than commit?

When T1 rollback, then T2 must rollback too!

- cascading rollback schedule

To avoid cascading rollback, use **strict 2PL**.

T1	T2
write_lock(A)	
read(A)	
write(A)	
	~~write_lock(A)~~
write_lock(B)	
read(B)	
write(B)	
commit	
unlock(A)	
unlock(B)	
	write_lock(A)
	read(A)
	write(A)
	write_lock(B)
	read(B)
	write(B)
	commit
	unlock(A)
	unlock(B)

Strict 2PL ensures both a serializable schedule and one that avoids cascading rollback.

Note the change in order of commit in strict 2PL

Looking the above – We have ended with a serial schedule!!

Why to work, just to end with a serial schedule ?

Remember, if the operations were not conflicting, there would be no issues on interleaving operations.

T1	T2
write_lock(A)	
read(A)	
write(A)	
	write_lock(C)
	write(C)
read_lock(B)	
	write_lock(Y)

Deadlock

When you introduce a locking protocol, deadlocks always become a possibility.

Problem 1:

Consider the following two transactions:

```
T1 :  read(A);
      read(B);
      if A = 0 then B := B + 1;
      write(B);

T2 :  read(B);
      read(A);
      if B = 0 then A := A + 1;
      write(A);
```

Add lock and unlock instructions to transactions T1 and T2 , so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

Problem 2:

Drwa the precedence graph for schedule involving T1,T2 and T3. Is it conflict serializable?

T1	**T2**	**T3**
read(C)		
write(C)		
		read(C)
	read(B)	
read(B)		
	write(B)	
		write(C)
	read(C)	
		write(A)
		write(B)
read(D)		
	write(C)	