Ceng352 - Written Assignment 3

Bilge Eren 2171585

Q1)

a)
Both T1 and T2 both Strict 2PL schedules because of they release their locks when transaction is completed.
We know that Strict 2PL allows only schedules whose precedence graph is acyclic and also The schedule is serializable iff the precedence graph is acyclic, therefore Strict 2PL ensures that the schedule is conflict serializable.
So that conflict serializability guaranteed for both T1 and T2.

b)
Yes, deadlock is possible at a situation such as when T1 waits for release of lock B which is locked/hold by T2 and at the same time T2 waits for release of lock A which is locked/hold by T1.
In wait-die deadlock prevention schema, younger transaction should be aborted so T2 would be aborted.

c)
Strict 2PL avoids cascading aborts because all exclusive mode locks taken by a transaction must be held until that transaction commits/aborts.

d)
In this scenario both T1 and T2 are 2PL scheduled because in every transaction, all lock requests must precede all unlock requests. And 2PL scheduling ensures that the schedule is conflict serializable.

e)
Since both transactions want to hold the same lock first. Deadlock is not possible. So T1 will locks the A and before releasing A, it will lock B as well and after release of lock A by T1, T2 could starts by locking A.

f)
It is possible since locks release before commit so depending on start order either of them could read or write uncommitted data, so if one of them aborted the other one should be aborted as well.

T1	X1(A)	R1(A)	W1(A)	X1(B)	U(A)			R1(B)	W1(B)	U(B)						
T2						X2(A)	X2(B) Denied				Granted R2(B)	R2(A)	W2(A)	U(A)	W2(B)	U(B)

In yellow parts T2 reads and writes uncommitted data so at the end or any of these yellow stages if the T1 aborted T2 should also be aborted.

Q2)

a.i. TS(T1)=1, TS(T2)=2, TS(T3)=3

Operation	A			B			C		
	RTS	WTS	C	RTS	WTS	C	RTS	WTS	C
R1(A)	1	0	TRUE	0	0	TRUE	0	0	TRUE
R2(B)	1	0	TRUE	2	0	TRUE	0	0	TRUE
R3(A)	3	0	TRUE	2	0	TRUE	0	0	TRUE
W1(A) - ABORT	3	0	TRUE	2	0	TRUE	0	0	TRUE
R2(C)	3	0	TRUE	2	0	TRUE	2	0	TRUE
W3(B)	3	0	TRUE	2	3	FALSE	2	0	TRUE
W2(C)	3	0	TRUE	2	3	FALSE	2	2	FALSE
C1 - REJECT	3	0	TRUE	2	3	FALSE	2	2	FALSE
R2(A)	3	0	TRUE	2	3	FALSE	2	2	FALSE
W3(C)	3	0	TRUE	2	3	FALSE	2	3	FALSE
C3	3	0	TRUE	2	3	FALSE	2	3	TRUE
W2(B) - IGNORE	3	0	TRUE	2	3	FALSE	2	3	TRUE
C2	3	0	TRUE	2	3	TRUE	2	3	TRUE

a.ii. TS(T1)=2, TS(T2)=3, TS(T3)=1

Operation	A			B			C		
	RTS	WTS	C	RTS	WTS	C	RTS	WTS	C
R1(A)	2	0	TRUE	0	0	TRUE	0	0	TRUE
R2(B)	2	0	TRUE	3	0	TRUE	0	0	TRUE
R3(A)	2	0	TRUE	3	0	TRUE	0	0	TRUE
W1(A)	2	2	FALSE	3	0	TRUE	0	0	TRUE
R2(C)	2	2	FALSE	3	0	TRUE	3	0	TRUE
W3(B) - ABORT	2	2	FALSE	3	0	TRUE	3	0	TRUE
W2(C)	2	2	FALSE	3	0	TRUE	3	3	FALSE
C1	2	2	TRUE	3	0	TRUE	3	3	FALSE
R2(A)	3	2	TRUE	3	0	TRUE	3	3	FALSE
W3(C) - REJECT	3	2	TRUE	3	0	TRUE	3	3	FALSE
C3 - REJECT	3	2	TRUE	3	0	TRUE	3	3	FALSE
W2(B)	3	2	TRUE	3	3	FALSE	3	3	FALSE
C2	3	2	TRUE	3	3	TRUE	3	3	TRUE

b. Commit bit is mainly used for avoiding cascading aborts and implies recoverable schedule also with the help of commit bit reading uncommitted writes of data precluded. As in a.ii R2(A) can read A since C1 occurs before it. So without commit bit dirty reads and cascading aborts could occur.

Q3) a.

LSN	transId	prevLSN	type	pageId	Log entry	undoNextLSN
1	T1	-	Update	P1	Write A (A->A1)	-
2	T2	-	Update	P1	Write B (B->B2)	-
3	T2	2	Update	P2	Write C (C->C3)	-
4	T2	3	Abort	-	-	-
5	-	-	-	-	begin_checkpoint	-
6	-	-	-	-	Xact Table: T1: 1 T2: 4 DPT: P1: 1 P2: 3 end_checkpoint	
7	T2	-	CLR	-	Undo T2 LSN:3	2
8	T2	-	CLR	-	Undo T2 LSN:2	-1
9	T2	8	END	-	-	-
10	T3	-	Update	P2	Write D(D->D10)	-
11	T1	2	Commit	-	-	-
12	T1	11	END	-	-	-
13	T4	-	Update	P1	Write A (A1->A13)	-
14	T3	10	Update	P1	Write B(B->B14)	-
15	T4	13	Commit	-	-	-

Dirty Page Table:

pageID	recLSN
P1	2
P2	3

Active Transaction Table:

transID	lastLSN	status
T3	14	Active
T4	15	Commit

Pages in memory:

	A	B	C	D	pageLSN
P1	**A13**	**B14**	**-**	**-**	**14**
P2	**-**	**-**	**C**	**D10**	**10**

b. When the system crashes and restarts, only the part of the log that was flushed to disk remains. Everything else (transactions table, dirty pages table, tail of the log) is gone.
So the analysis phase is start after checkpoint. Which is end with LSN 6.

Dirty Page Table:

pageID	recLSN
P1	2
P2	3

Active Transaction Table:

transID	lastLSN	status
T3	14	Active
T4	15	Commit

c. The REDO phase starts at the firstLSN, which is the smallest LSN in the Dirty Page Table. In this example, it's LSN 1. Scan forward through the log starting at LSN 2.
LSN 2: Read P1, If PageLSN<2, redo LSN 2 -> **DON'T**
LSN 3: Read P2, If PageLSN<3, redo LSN 3 -> **DON'T**
LSN 4: **REDONE**
LSN 5: **SKIP**
LSN 6: **SKIP**
LSN 7: **REDONE**
LSN 8: **REDONE**
LSN 9: **SKIP**
LSN 10: Read P2, If PageLSN<10, redo LSN 10 -> **REDONE - Set P2.pageLSN = 10**
LSN 11: **SKIP**
LSN 12: **SKIP**
LSN 13: Read P1, If PageLSN<10, redo LSN 10 -> **REDONE - Set P1.pageLSN = 13**
LSN 14: Read P1, If PageLSN<10, redo LSN 10 -> **REDONE - Set P1.pageLSN = 14**
LSN 15: **SKIP**

Dirty Page Table:

pageID	recLSN
P1	2
P2	3

Active Transaction Table:

transID	lastLSN	status
T3	14	Active
T4	15	Commit

Pages in memory:

	A	B	C	D	pageLSN
P1	**A13**	**B14**	**-**	**-**	**14**
P2	**-**	**-**	**C**	**D10**	**10**

d. ToUndo = {14,10}

LSN	transId	prevLSN	type	pageId	Log entry	undoNextLSN
1	T1	-	Update	P1	Write A (A->A1)	-
2	T2	-	Update	P1	Write B (B->B2)	-
3	T2	2	Update	P2	Write C (C->C3)	-
4	T2	3	Abort	-	-	-
5	-	-	-	-	begin_checkpoint	-
6	-	-	-	-	Xact Table: T1: 1 T2: 4 DPT: P1: 1 P2: 3 end_checkpoint	
7	T2	-	CLR	-	Undo T2 LSN:3	2
8	T2	-	CLR	-	Undo T2 LSN:2	-1
9	T2	8	END	-	-	-
10	T3	-	Update	P2	Write D(D->D10)	-
11	T1	2	Commit	-	-	-
12	T1	11	END	-	-	-
13	T4	-	Update	P1	Write A (A1->A13)	-
14	T3	10	Update	P1	Write B(B->B14)	-
15	T4	13	Commit	-	-	-
16	T4	15	END	-	-	-
17	T3	-	CLR	-	Undo T3 LSN:14	10
18	T3	-	CLR	-	Undo T3 LSN:10	-1
19	T3	18	END	-	-	-

Pages in memory:

	A	B	C	D	pageLSN
P1	**A13**	**B**	**-**	**-**	**17**
P2	**-**	**-**	**C**	**D**	**18**