CS143: Homework 6

Problem A: T1 is the oldest transaction, and T3 is the youngest.

T1	T2	T3
write D		
	write C	
read C		
		read C
		write A
	read A	
write A		

1. Is this first schedule conflict-serializable[1]?

Answer: We have that T2 $\rightarrow$ T3 on C and T3 $\rightarrow$ T2 on A. The directed cycle T3 $\leftrightarrows$T2 tells us that the schedule is not conflict-serializable.

Now assume that the transaction manager uses a 2PL protocol where each exclusive/shared lock is set just before it is needed for the write/read action.

2. If we do not use any deadlock prevention strategy, will the resulting transactions (i) complete, or (ii) deadlock[1]? If your answer is (i), show a completed schedule;

Answer: if it is (ii) show the schedule up to the deadlock. So T2 sets a lock-X C just before write C, and because of 2PL it must keep until it is done with its read A. Thus when T1 does lock-S C it stops with a wait-for arc T1 $\rightarrow$ T2. Ditto for T3 that sets the wait-for arc T3 $\rightarrow$ T2. There is no cycle: thus no deadlock. T2 completes. Then T3 and T1 will complete also.

3. If we use a wait-die deadlock prevention strategy will the resulting transactions (i) complete, or (ii) deadlock[1]? If your answer is (i), show a completed schedule; if it is (ii) show the schedule up to the deadlock.

Answer: Obviously there is going to be no deadlock since Wait-Die prevents deadlocks. Here, T1 will wait for the older transaction to release C. But T3 is younger than T2; so when it requests C it will actually die. Again a correct completion sequence is T2, T1, T3.

From now on consider the second schedule below and use that in your answer:

T1	T2	T3
write D		
	write C	
read C		
		write A
		read C
	read A	
write A		

[1] Justify your answer using the applicable graph

4. Is this second schedule conflict-serializable[1]?

Answer: Here too, we have that T2 $\rightarrow$ T3 on C and T3 $\rightarrow$ T2 on A. This cycles tells us that the schedule is not confict-serializable.

Now assume that the transaction manager uses a 2PL protocol where each exclusive/shared lock is set just before it is needed for the write/read action, and answer the following questions for this second schedule.

5. If we do not use any deadlock prevention strategy, will the resulting transactions (i) complete, or (ii) deadlock[1]? If your answer is (i), show a completed schedule; if it is (ii) show the schedule up to the deadlock.

Answer: So T2 sets a lock-X C just before write C, and because of 2PL it must keep until it is done with its read A. Thus when T1 does lock-S C it stops with a wait-for arc T1 $\rightarrow$ T2. Next, T3 does lock-X A, and then by requesting lock-s C it sets a wait-for arc T3 $\rightarrow$ T2. Thua, so far, no cycles and no deadlock. But then T2 does lock-S A and sets a wait-for arc: T2 $\rightarrow$ T3. Thus we now have deadlock as per the directed cycle T3 $\leftrightarrows$ T2 in the wait-for graph.

6. If we use a Wound-Wait deadlock prevention strategy will the resulting transactions (i) complete, or (ii) deadlock[1]? If your answer is (i), show a completed schedule; if it is (ii) show the schedule up to the deadlock.

Answer: Obviously there is going to be no deadlock since Wound-Wait prevents deadlocks. Here, T1 is the older transaction requesting C held by the younger T2. Thus T1 wounds T2 which dies releasing C. Then T3 locks A and C and completes releasing them both. Finally T2 restarts and completes. Thus the correct completion sequence is T3, T1 and finally T2.

Problem B 1. Consider the following schedule: (w3(A) means that transaction T3 writes A, C3: T3 commits):

w3(A) r1(A) c3 w1(B) c1 r2(B) w2(C) r4(B) c2c4

(a) Is it a serial schedule?

(b) Is the schedule conflict serializable? If so, what are all the equivalent serial schedules?

(c) Is the schedule recoverable? If not, can we make it recoverable by moving a single commit operation to a different position?

(d) Is the schedule cascadeless? If not, can we make it cascadeless by moving a single commit operation to a different position?

T1	T2	T3	T4
		w(A)	
r(A)			
		commit	
w1(B)			
commit			
	r(B)		
	w(C)		
			r(B)
	commit		
			commit

(a) this is not a serial schedule

(b) T3 → T1 (due to A) T1→T2 (do to B) T1→T4 (due to B)

No cycle. The schedule is serializable. e.g. T3 before T1 before T2 and before T4

(c) Dirty data with current schedule: A and B. A is written by T3 and read by T1 after T1 commits. Then, the recovery process will undo w1(B) and generate a schedule that could not have been generated by an equivalent serial schedule (unrecoverability of schedules).

d) Dirty data with current schedule: A and B. A is written by T3 and read by T1 after T1 commits—no cascading of rollback is needed because of A. However, B is read by T1 and T4 before they commit: thus we have a cascade of rollbacks if T1 fails. To make this schedule recoverable and cascadeless, we must delay the commit of T1 until T2 and T4 commit.