CS5200 Hwk10
Yixiao Chen 002198256

Q1:
Given the following 2 different transactions. List all potential schedules for T1 and T2 and determine which schedules are conflict serializable and which are not. (10 POINTS)

Schedule 1: conflict serializable

Time	Transaction1	Transaction2
1	Begin transaction	
2	READ(X)	
3	X = X – N;	
4	WRITE(X)	
5	READ(Y)	
6	Y = Y + N;	
7	WRITE(Y)	
8	Commit	Begin transaction
9		READ(X)
10		X = X + M;
11		WRITE(X)
12		Commit

Schedule 2: conflict serializable

Time	Transaction1	Transaction2
1		Begin transaction
2		READ(X)
3		X = X + M;
4		WRITE(X)
5	Begin transaction	Commit
6	READ(X)	
7	X = X – N;	
8	WRITE(X)	
9	READ(Y)	
10	Y = Y + N;	
11	WRITE(Y)	
12	Commit	

Schedule 3: conflict serializable

Time	Transaction1	Transaction2
1	Begin transaction	
2	READ(X)	
3	X = X – N;	
4	WRITE(X)	Begin transaction
5		READ(X)
6		X = X + M;
7		WRITE(X)
8		Commit
9	READ(Y)	
10	Y = Y + N;	
11	WRITE(Y)	
12	Commit	

Schedule 4: not conflict serializable

Time	Transaction1	Transaction2
1	Begin transaction	
2	READ(X)	
3	X = X – N;	Begin transaction
4		READ(X)
5		X = X + M;
6		WRITE(X)
7		Commit
8	WRITE(X)	
9	READ(Y)	
10	Y = Y + N;	
11	WRITE(Y)	
12	Commit	

Schedule 5: not conflict serializable

Time	Transaction1	Transaction2
1	Begin transaction	
2	READ(X)	
3	X = X – N;	Begin transaction
4		READ(X)
5	WRITE(X)	
6		X = X + M;
7		WRITE(X)
8		Commit
9	READ(Y)	
10	Y = Y + N;	
11	WRITE(Y)	
12	Commit	

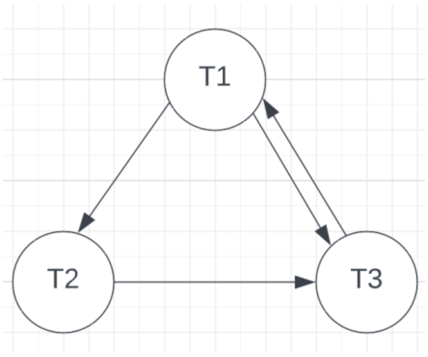
Q2 & Q3:

Which of the following schedules is conflict serializable? For each serializable schedule determine the equivalent serial schedule. (10 POINTS)

Draw the precedence graph for the 3 schedules in problem 2. (20 POINTS)

Schedule1: not conflict serializable

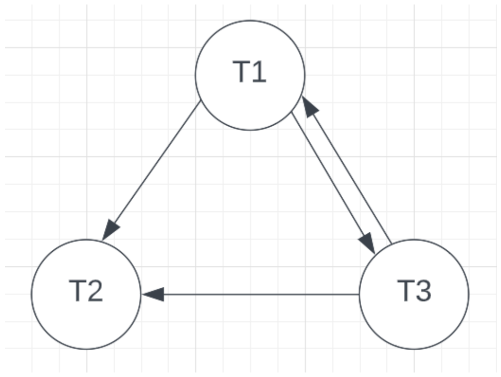
Time	T1	T2	T3
1	READ(X)		
2			READ(X)
3	WRITE(X)		
4		READ(X)	
5			WRITE(X)



The precedence graph **contains a circle**. Therefore, this is not a conflict serializable schedule.

Schedule2: not conflict serializable

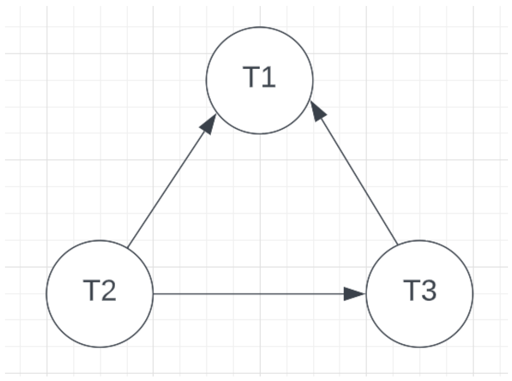
Time	T1	T2	T3
1	READ(X)		
2			READ(X)
3			WRITE(X)
4	WRITE(X)		
5		READ(X)	



The precedence graph **contains a circle**. Therefore, this is not a conflict serializable schedule.

Schedule3: conflict serializable

Time	T1	T2	T3
1			READ(X)
2		READ(X)	
3			WRITE(X)
4	READ(X)		
5	WRITE(X)		



Equivalent serial schedule: T2, T3, T1

Time	T1	T2	T3
1		READ(X)	
2			READ(X)
3			WRITE(X)
4	READ(X)		
5	WRITE(X)		

Q4:

Provide a schedule that exhibits the deadlock problem. Describe the issue. (10 POINTS)

Time	T1	T2
1	Begin T1	
2	write_lock(X)	Begin T2
3	Read(X)	Write_lock(Y)
4	X = X – 10;	Read(Y)
5	WRITE(X)	Y = Y + 100;
6	write_lock(Y)	WRITE(Y)
7	WAIT	Write_lock(X)
8	WAIT	WAIT
9	WAIT	WAIT
10	…	WAIT
11	…	…
12	…	…

Deadlock is an impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.
As shown in the above schedule, T1 is waiting for the lock on Y held by T2 while T2 is also waiting for T1 to release the lock on X; Both transactions are stuck at this position and none of them could move forward.

Q5:

Apply the timestamping algorithm to the following schedule. State if it can be performed as is or what transactions will need to be restarted given the basic timestamping ordering algorithm. (20 POINTS)

Time	Operation	Timestamp modification
1	B READ(Z)	Ts(B) = 1 Read_timestamp(Z) = 1
2	B READ(Y)	Read_timestamp(Y) = 1
3	B WRITE(Y)	Write_timestamp(Y) = 1
4	C READ(Y)	Ts(C) = 4 Read_timestamp(Y) = 4
5	C READ(Z)	Read_timestamp(Z) = 4
6	A READ(X)	Ts(A) = 6 Read_timestamp(X) = 6
7	A WRITE(X)	Write_timestamp(X) = 6
8	C WRITE(Y)	Write_timestamp(Y) = 4
9	C WRITE(Z)	Write_timestamp(Z) = 4
10	B READ(X)	Ts(B) =1 < Read_timestamp(X) = 6
11	A READ(Y)	Read_timestamp(Y) = 6
12	A WRITE(Y)	Wite_timestamp(Y) = 6
13	B READ(Z)	Ts(B) = 13 Read_timestamp(Z) = 13
14	B READ(Y)	Read_timestamp(Y) = 13
15	B WRITE(Y)	Wite_timestamp(Y) = 13
16	B READ(X)	Read_timestamp(X) = 13
17	B WRITE(X)	Wite_timestamp(X) = 13

When time = 10, transaction B requested to read X, while Ts(B) < read timestamp of X. Therefore, Transaction B needs to abort and restarted with a later timestamp (Ts(B) = 13)

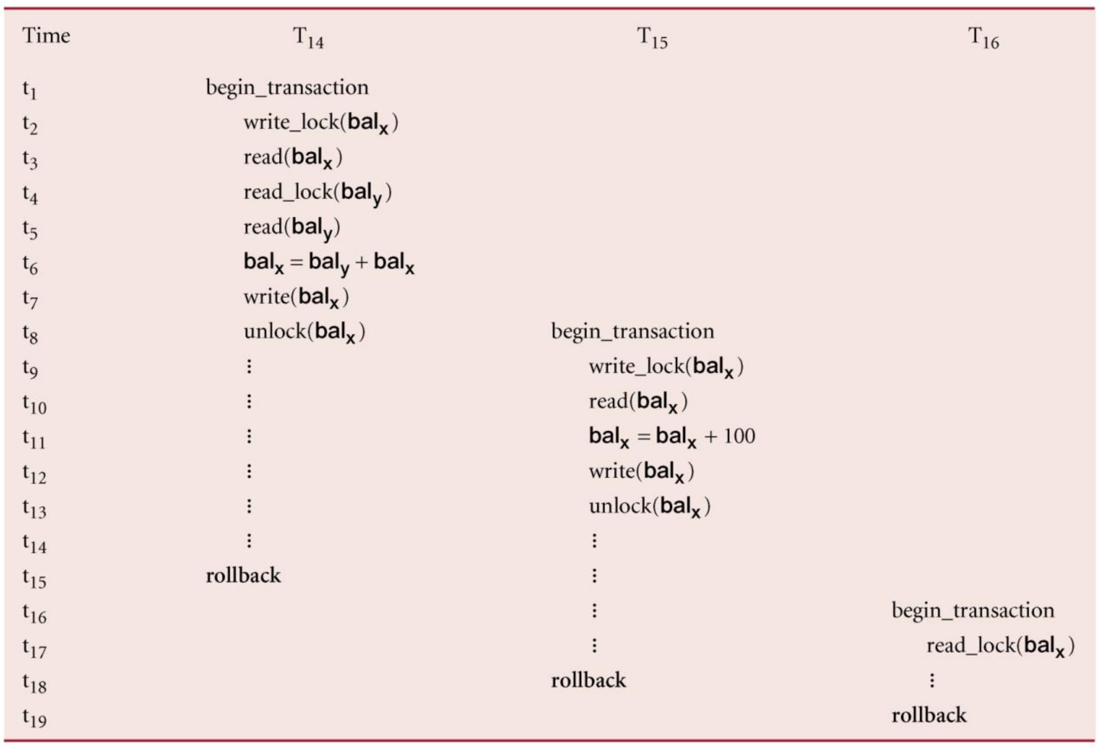
Q6:
Below is a log corresponding to a particular schedule at the point of a system crash for 4 transactions T1, T2, T3, and T4. Suppose that we use the immediate update protocol with checkpointing. Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations in the log are redone and which operations in the log are undone. State whether any cascading rollbacks take place. (20 POINTS)

Noticed that there is a **CHECKPOINT** in the log file.
Since **Transaction 1** has already **COMITTED** before the checkpoint so we could be sure that the changes have already been written to the secondary storage.
While for the rest of the transactions:
Noticed that there is a **COMMIT** for **Transaction 4** before the system crash, so all operations of **T4** needs to be **redo**.
Since there aren't and COMMIT in sight for T2 and T3, that means the operations presented as **T2** and **T3** need to be **undone**. [T2, T3 needs to **rolled back** (undo the actions/effects that had not committed start from the bottom/ according to a reverse order.)]

There is **no cascade rollback** in this case.

Q7:
Describe what a cascading rollback is. (10 points)

Cascading rollback is when a single transaction failure leads to a series of transaction rollbacks
For example, as shown in the screenshot below. If T14 aborts. Since T15 is dependent on T14, T15 must also be rolled back. Since T16 is dependent on T15, it too must be rolled back.
This is called cascading rollback



Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock(**bal_x**)		
t_3	read(**bal_x**)		
t_4	read_lock(**bal_y**)		
t_5	read(**bal_y**)		
t_6	$bal_x = bal_y + bal_x$		
t_7	write(**bal_x**)		
t_8	unlock(**bal_x**)	begin_transaction	
t_9	⋮	write_lock(**bal_x**)	
t_{10}	⋮	read(**bal_x**)	
t_{11}	⋮	$bal_x = bal_x + 100$	
t_{12}	⋮	write(**bal_x**)	
t_{13}	⋮	unlock(**bal_x**)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock(**bal_x**)
t_{18}		rollback	⋮
t_{19}			rollback