COS 425, Fall 2004
Solutions to Problem Set 6

(1)

Part a.

Sequence S1, with Strict 2PL and wait-die policy for deadlock prevention:

T1	T2	T3
S(X)		
R(X)		
	X(X) – aborted as T2 has lower priority than T1	
		X(Y)
		W(Y)
X(Y) – must wait because T1 has higher priority than T3		
		Commit
W(Y)		
Commit		
	X(X)	
	W(X)	
	X(Y)	
	W(Y)	
	Commit	

NOTE: T2 is aborted as a result of the wait-die policy. The exact time of when it is restarted may depend on the details of the transaction scheduler. In the execution shown above, T2 is restarted after T1 has finished. It may happen that the scheduler restarts T2 before T1 has finished and while T1 is still holding the lock on X. In this case, when T2 asks for a lock on X, it may get aborted again. The precise execution details, therefore, depend on how the scheduler works. (A similar comment applies to the rest of the transaction execution schedules in this problem.)

Sequence S2, with Strict 2PL and wait-die policy for deadlock prevention:

T1	***T2***	***T3***
S(X)		
R(X)		
	X(Y)	
	W(Y)	
	X(X) – aborted as T2 has lower priority that T1	
		X(Y)
		W(Y)
X(Y) – must wait because T1 has higher priority than T3		
		Commit
W(Y)		
Commit		
	X(Y)	
	W(Y)	
	X(X)	
	W(X)	
	Commit	

Part b.

Sequence S1, with Strict 2PL and wound-wait policy for deadlock prevention:

T1	***T2***	**T3**
S(X)		
R(X)		
	X(X) – must wait because T2 has lower priority than T1	
		X(Y)
		W(Y)
X(Y) – must abort T3 because T1 has higher priority than T3		
		Abort
W(Y)		
Commit		
	W(X)	
	X(Y)	
	W(Y)	
	Commit	
		X(Y)
		W(Y)
		Commit

Sequence S2, with Strict 2PL and wound-wait policy for deadlock prevention:

T1	***T2***	***T3***
S(X)		
R(X)		
	X(Y)	
	W(Y)	
	X(X) – must wait because T2 has lower priority than T1	
		X(Y) – must wait because T3 has lower priority than T2
X(Y) – must abort T2 because T1 has higher priority than T2		
	Abort	
W(Y)		
Commit		
		W(Y)
		Commit
	X(Y)	
	W(Y)	
	X(X)	
	W(X)	
	Commit	

Part c.

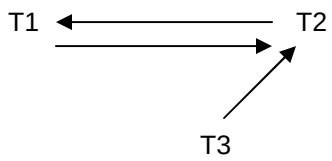
Sequence S1, with strict 2PL and deadlock detection (no prevention):

T1	T2	T3
S(X)		
R(X)		
	X(X) – must wait for T1 to release lock	
		X(Y)
		W(Y)
X(Y) – must wait for T3 to release lock		
		Commit
W(Y)		
Commit		
	X(X)	
	W(X)	
	X(Y)	
	W(Y)	
	Commit	

Sequence S2, with strict 2PL and deadlock detection (no prevention):

T1	*T2*	*T3*
S(X)		
R(X)		
	X(Y)	
	W(Y)	
	X(X) – must wait for T1 to release lock	
		X(Y) – must wait for T2 to release lock
X(Y) – must wait for T2 to release lock.. DEADLOCK!!		

The waits-for graph in this case is:



(2)

Part a: The following schedule is not view serializable, recoverable, and does not avoid cascading aborts:

T1	**T2**
R(X)	
	R(X)
	W(X)
R(X)	
W(X)	
	Commit
Commit	

- The schedule is not view serializable because both T1, T2 read the initial value of X and both write to X, so in both the possible serial schedules the second transaction will not read the initial value of X.
- The schedule is recoverable because T1 reads the value of X written by T2 and commits after T2 does.
- The schedule does not avoid cascading aborts since T1 reads the X value modified by T2 before it is committed. So if T2 is aborted, then T1 must be aborted.

Part b: The following schedule avoids cascading aborts, but is not strict, and is conflict serializable:

T1	**T2**
W(X)	
	W(X)
	Commit
Commit	

- The schedule avoids cascading aborts because neither T1 nor T2 reads the value of X written by the other.
- The schedule is not strict because the value of X written by T1 is overwritten by T2 before T1 commits.
- The schedule is conflict serializable because it is conflict-equivalent to the serial schedule T1 followed by T2 (i.e. T1's commit can be pushed before the **W**(X) of T2).

Part c: The following schedule is strict, view serializable but not conflict serializable:

T1	***T2***	***T3***
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit

- The schedule is strict: all writes are immediately followed by commits.
- The schedule is view equivalent to the serial schedule that performs T1, then T2, then T3. So it is view serializable.
- The schedule is not conflict serializable since all the read and write operations conflict and hence cannot be reordered and obviously this schedule itself is not a serial schedule.

(3)

1.
The extended figure is shown below. Note that only CLR entries have the undoNextLSN field.

LSN	PrevLSN	undoNextLSN	LOG
00	Null	-	Update: T1 writes P2
10	00	-	Update: T1 writes P1
20	Null	-	Update: T2 writesP5
30	Null	-	Update: T3 writes P3
40	30	-	T3 commit
50	20	-	Update: T2 writes P5
60	50	-	Update: T2 writes P3
70	60	-	T2 abort

2.
The actions taken to rollback transaction T2 are (recall that undo operations are performed in backward order of time, opposite of redo):

1. The lastLSN for T2 is 70, since it's an abort, the prevLSN 60 is fetched.

2. LSN 60 is an update operation for P3 with PrevLSN 50. First, a CLR record is written with undoNextLSN 50, and then the update of P3 is undone. Next, LSN 50 is fetched.
3. LSN 50 is an update operation for P5 with PrevLSN 20. First, a CLR record is written with undoNextLSN 20, and then the update of P5 is undone. Next LSN 20 is fetched.
4. LSN 20 is an update operation for P5 with PrevLSN equal to Null. First, a CLR record is written with undoNextLSN equal to Null, and then the update of P5 is undone.
5. An “end” entry is written

3.
The log after T2 is rolled back:

LSN	PrevLSN	undoNextLSN	LOG
00	Null	-	Update: T1 writes P2
10	00	-	Update: T1 writes P1
20	Null	-	Update: T2 writesP5
30	Null	-	Update: T3 writes P3
40	30	-	T3 commit
50	20	-	Update: T2 writes P5
60	50	-	Update: T2 writes P3
70	60	-	T2 abort
80	70	50	CLR: Undo T2 LSN 60
90	80	20	CLR: Undo T2 LSN 50
100	90	Null	CLR: Undo T2 LSN 20
110	100	-	T2 end

(4)

Part a:
The transaction table:

Transaction ID	lastLSN	Status
T1	0	In progress
T2	1	In progress

The dirty page table:

pageID	RecLSN
Pg1	0
Pg2	1

Part b:
In the following analysis, it is assumed that the log through LSN 9 was written to stable storage before the first crash and the log through LSN 11 was written to stable storage before the second crash. Otherwise, the analysis would be on the portions of the log that have been saved to stable storage before each crash and thus survive the crashes.

Actions of the first crash recovery:

- **Analysis phase:** The recovery manager first builds the transaction table and dirty page table as were recorded in the "end checkpoint" entry (as in part a). Then the analysis phases scans the rest of the log after the "end checkpoint" entry, and adds transaction T3 to the transaction table and removes transaction T1 (as it has an "end" entry and is no longer active). Transaction T2 has committed but has no "end" record, so its status is changed to "committed". In addition, pages 3,4,5 are added to the dirty page table. Also, the smallest recLSN in the dirty page table is 0, so the redo phase will start from this LSN.
- **Redo phase:** The redo phase begins with log record 0, which is the minimum recLSN in the dirty page table, and reapplies all redoable actions (only update actions in this case). If some pages have already been written to disk, then their pageLSN entries will reflect this and updates for these pages need not be redone. Since T2 commits, an end record is written for it at LSN 10 and T2 is removed from the transaction table.

LSN	PrevLSN	UndoNextLSN	LOG
10	9	-	T2 end

- **Undo phase:** The initial ToUndo set consists of LSN 6 (the lastLSN field for T3 in the transaction table, and T3 is the only loser transaction as T2 already committed and need not be undone). Hence, the following CLR is written to the log:

LSN	PrevLSN	UndoNextLSN	LOG
11	6	4	CLR: Undo T3 LSN 6

Next, there is another crash. The actions of the second crash recovery:

- **Analysis phase:** The recovery manager first builds the transaction table and dirty page table as were recorded in the "end checkpoint" entry (as in part a). Then the analysis phases scans the rest of the log, and adds transaction T3 to the transaction table and removes transaction T1, T2 (as now both have an "end" entry and are no longer active). In addition, pages 3,4,5 are added to the dirty page table.

- **Redo phase:** The redo phase begins with log record 0, which is the minimum recLSN in the dirty page table, and reapplies all redoable actions (including the undo action of T2 this time). If some of the changes made during the previous redo were actually written to disk, the pageLSNs on the affected pages are used to detect this situation and avoid writing these pages again.

- **Undo phase:** The initial ToUndo set consists of LSN 11 (the lastLSN field for T3 in the current transaction table). It is processed by replacing this value with the undoNextLSN value, which is 4. Then, the following record is written to log which is for the undoing of LSN 4, and T3 can now be ended since the prevLSN value of LSN 4 is null.

LSN	PrevLSN	UndoNextLSN	LOG
12	11	Null	CLR: Undo T3 LSN 4
13	12	-	T3 end