Chap 19 solutions

19.1.1

The difference between strict and nonstrict locking for this example is only in the matter of whether the lock on *A* is released prior to the write of *B*. That is, the lock on *A* must be taken before $r_1(A)$. The lock on *B* can occur in any of the three positions prior to $r_1(B)$ (3 choices).

If locking is strict, then the two unlocks must occur, in either order, after the second write action. Thus, there are 6 orders for strict, 2-phase locking.

If locking is not strict, the unlock of *B* must occur after $w_1(B)$, but then the unlock of *A* can occur in any of the three positions after $w_1(A)$. Thus, there are 9 orders if strictness is not required, and 3 of these must be nonstrict.

19.1.2a)

T_1 wrote only *B*, but that value was later read by T_3. Thus, T_3 must be rolled back. T_3 wrote *D*, but no transaction has read *D*, so no further rollbacks are needed.

19.1.2b)

T_2 rolled back since it reads B written by T_1.
T_3 rolled back since it reads C written by T_2.

19.1.2c)

T_2 rolled back since it reads B written by T_1.
T_3 rolled back since it reads B written by T_1.

19.1.2d)

T_3 rolled back as it reads B written by T_1.

19.1.3a)

Inserting commit actions we get the following schedule:
$r_1(A)r_2(B)w_1(B)\mathbf{c_1}w_2(C)\mathbf{c_2}r_3(B)r_3(C)w_3(D)\mathbf{c_3}$

i) If the lost tail of log begins before c_1, T_1, T_2 and T_3 are considered uncommitted. If lost tail starts between c_1 and c_2, T_2 and T_3 considered uncommitted. If lost tail starts between c_2 and c_3, T_3 considered uncommitted.
ii) No dirty reads
iii) If c_1 lost, dirty read of B by T_3 since it is written by T_1.

19.1.3b)

$r_1(A)w_1(B)\mathbf{c_1}r_2(B)w_2(C)\mathbf{c_2}r_3(C)w_3(D)\mathbf{c_3}$

i) If lost tail begins before c_1, T_1, T_2 and T_3 considered uncommitted. If lost tail starts between c_1 and c_2, T_2 and T_3 considered uncommitted. If lost tail between c_2 and c_3, t_3 considered uncommitted.
ii) No dirty reads
iii) If c_1 lost, dirty read of B by T_2 since it is written by T_1. If c_2 lost dirty read of C by T_3 since it is written by T_2.

19.1.3c)

$r_2(A)r_3(A)r_1(A)w_1(B)\mathbf{c_1}r_2(B)r_3(B)w_2(C)\mathbf{c_2}r_3(C)\mathbf{c_3}$

i) If lost tail begins before c1, T1, T2 and T3 considered uncommitted. If lost tail begins between c1 and c2, T2 and T3 considered uncommitted. If lost tail begins between c2 and c3, T3 considered uncommitted.
ii) No dirty reads
iii) If lost record is c1, dirty read of B by T2 since it is written by T1. If lost record if c2, then dirty read of C by T3 since it is written by T2.

19.1.3d)

$r_2(A)r_3(A)r_1(A)w_1(B)\mathbf{c_1}r_3(B)w_2(C)\mathbf{c_2}r_3(C)\mathbf{c_3}$

i) If lost tail before c_1, T_1, T_2 and T_3 considered uncommitted. If lost tail between c_1 and c_2, T_2 and T_3 considered uncommitted. If lost tail between c_2 and c_3, T_3 considered uncommitted.
ii) No dirty reads
iii) If c_1 lost, dirty read of B by T_2 since it is written by T_1. If c_2 lost, dirty read of C by T_3 since it is written by T_2.

19.1.4a)

For Recoverable schedules, each transaction must commit only after the transaction it read from commits. There is one read in each transaction.
Therefore the constraints are
(i) When r1(C) follows w2(C) , c1 must follow c2
(ii) When r2(B) follows w1(B) , c2 must follow c1

There are 10 transactions that violate (i) viz.

w2(A) r2(B) w2(C) w1(A) w1(B) r1(C) c_1 c_2
w2(A) r2(B) w1(A) w2(C) w1(B) r1(C) c_1 c_2
w2(A) r2(B) w1(A) w1(B) w2(C) r1(C) c_1 c_2
w2(A) w1(A) r2(B) w2(C) w1(B) r1(C) c_1 c_2
w2(A) w1(A) r2(B) w1(B) w2(C) r1(C) c_1 c_2
w2(A) w1(A) w1(B) r2(B) w2(C) r1(C) c_1 c_2
w1(A) w2(A) r2(B) w2(C) w1(B) r1(C) c_1 c_2
w1(A) w2(A) r2(B) w1(B) w2(C) r1(C) c_1 c_2
w1(A) w2(A) w1(B) r2(B) w2(C) r1(C) c_1 c_2
w1(A) w1(B) w2(A) r2(B) w2(C) r1(C) c_1 c_2

There are 13 transactions that violate (ii) viz.

w2(A) w1(A) w1(B) r2(B) w2(C) c_2 r1(C) c_1
w2(A) w1(A) w1(B) r2(B) w2(C) r1(C) c_2 c_1
w2(A) w1(A) w1(B) r2(B) r1(C) w2(C) c_2 c_1
w2(A) w1(A) w1(B) r1(C) r2(B) w2(C) c_2 c_1
w1(A) w2(A) w1(B) r2(B) w2(C) c_2 r1(C) c_1
w1(A) w2(A) w1(B) r2(B) w2(C) r1(C) c_2 c_1
w1(A) w2(A) w1(B) r2(B) r1(C) w2(C) c_2 c_1
w1(A) w2(A) w1(B) r1(C) r2(B) w2(C) c_2 c_1
w1(A) w1(B) w2(A) r2(B) w2(C) c_2 r1(C) c_1
w1(A) w1(B) w2(A) r2(B) w2(C) r1(C) c_2 c_1
w1(A) w1(B) w2(A) r2(B) r1(C) w2(C) c_2 c_1
w1(A) w1(B) w2(A) r1(C) r2(B) w2(C) c_2 c_1
w1(A) w1(B) r1(C) w2(A) r2(B) w2(C) c_2 c_1

Total possible orders are (8 choose 4) = 70.
Out of these, 23 (13+10) are not recoverable.
Thus 47 (70 - 23) are recoverable.

<u>19.1.4b)</u>
In an ACR schedule, transactions only read values written by committed transactions.
Again each transaction has one read.
Therefore the constraints are
(i) When r1(C) follows w2(c), r1(c) must also follow c2
(ii) When r2(B) follows w1(B), r2(B) must also follow c1

18 schedules that violate (i) only are
w2(A) r2(B) w2(C) w1(A) w1(B) r1(C) c_2 c_1
w2(A) r2(B) w2(C) w1(A) w1(B) r1(C) c_1 c_2
w2(A) r2(B) w1(A) w2(C) w1(B) r1(C) c_2 c_1
w2(A) r2(B) w1(A) w2(C) w1(B) r1(C) c_1 c_2
w2(A) r2(B) w1(A) w1(B) w2(C) r1(C) c_2 c_1
w2(A) r2(B) w1(A) w1(B) w2(C) r1(C) c_1 c_2
w2(A) w1(A) r2(B) w2(C) w1(B) r1(C) c_2 c_1
w2(A) w1(A) r2(B) w2(C) w1(B) r1(C) c_1 c_2
w2(A) w1(A) r2(B) w1(B) w2(C) r1(C) c_2 c_1
w2(A) w1(A) r2(B) w1(B) w2(C) r1(C) c_1 c_2
w1(A) w2(A) r2(B) w2(C) w1(B) r1(C) c_2 c_1
w1(A) w2(A) r2(B) w2(C) w1(B) r1(C) c_1 c_2
w1(A) w2(A) r2(B) w1(B) w2(C) r1(C) c_2 c_1
w1(A) w2(A) r2(B) w1(B) w2(C) r1(C) c_1 c_2
w1(A) w2(A) w1(B) r2(B) w2(C) r1(C) c_2 c_1
w1(A) w2(A) w1(B) r2(B) w2(C) r1(C) c_1 c_2
w1(A) w1(B) w2(A) r2(B) w2(C) r1(C) c_2 c_1

w1(A) w1(B) w2(A) r2(B) w2(C) r1(C) c_1   c_2

9 schedules that violate only (ii) are
w2(A) w1(A) w1(B) r2(B) w2(C) c_2 r1(C) c_1
w2(A) w1(A) w1(B) r2(B) r1(C) w2(C) c_2   c_1
w2(A) w1(A) w1(B) r2(B) r1(C) w2(C) c_1   c_2
w2(A) w1(A) w1(B) r2(B) r1(C) c_1 w2(C) c_2
w2(A) w1(A) w1(B) r1(C) r2(B) w2(C) c_2   c_1
w2(A) w1(A) w1(B) r1(C) r2(B) w2(C) c_1   c_2
w2(A) w1(A) w1(B) r1(C) r2(B) c_1 w2(C) c_2
w1(A) w2(A) w1(B) r2(B) w2(C) c_2 r1(C) c_1
w1(A) w1(B) w2(A) r2(B) w2(C) c_2 r1(C) c_1

Also schedules that cause violations of both (i) and (ii) are
w2(A) w1(A) w1(B) r2(B) w2(C) r1(C) c_2   c_1
w2(A) w1(A) w1(B) r2(B) w2(C) r1(C) c_1   c_2

Thus there are 41 ACR schedules (70 - 18 - 9 - 2).

19.1.4c)
All corresponding actions of the two transactions on same element are possible comflicts i.e. cannot be swapped without affcting the serializale nature.

Of the 47 recoverable schedules, below 19 are not serializable due to conflicts.
w2(A) r2(B) w1(A) w1(B) r1(C) w2(C) c_2   c_1
w2(A) r2(B) w1(A) w1(B) r1(C) w2(C) c_1   c_2
w2(A) r2(B) w1(A) w1(B) r1(C) c_1 w2(C) c_2
w2(A) w1(A) r2(B) w1(B) r1(C) w2(C) c_2   c_1
w2(A) w1(A) r2(B) w1(B) r1(C) w2(C) c_1   c_2
w2(A) w1(A) r2(B) w1(B) r1(C) c_1 w2(C) c_2
w2(A) w1(A) w1(B) r2(B) r1(C) w2(C) c_1   c_2 .
w2(A) w1(A) w1(B) r2(B) r1(C) c_1 w2(C) c_2 .
w2(A) w1(A) w1(B) r1(C) r2(B) w2(C) c_1   c_2 .
w2(A) w1(A) w1(B) r1(C) r2(B) c_1 w2(C) c_2 .
w2(A) w1(A) w1(B) r1(C) c_1 r2(B) w2(C) c_2
w1(A) w2(A) r2(B) w2(C) c_2 w1(B) r1(C) c_1
w1(A) w2(A) r2(B) w2(C) w1(B) c_2 r1(C) c_1
w1(A) w2(A) r2(B) w2(C) w1(B) r1(C) c_2   c_1 .
w1(A) w2(A) r2(B) w1(B) w2(C) c_2 r1(C) c_1
w1(A) w2(A) r2(B) w1(B) w2(C) r1(C) c_2   c_1 .
w1(A) w2(A) r2(B) w1(B) r1(C) w2(C) c_2   c_1
w1(A) w2(A) r2(B) w1(B) r1(C) w2(C) c_1   c_2
w1(A) w2(A) r2(B) w1(B) r1(C) c_1 w2(C) c_2

Thus 28 transactions (47 - 19) are both recoverable and serializable.

19.1.4d)

Of the 41 ACR schedules, following 13 are not conflict serializable.
w2(A) r2(B) w1(A) w1(B) r1(C) w2(C) c_2 c_1
w2(A) r2(B) w1(A) w1(B) r1(C) w2(C) c_1 c_2
w2(A) r2(B) w1(A) w1(B) r1(C) c_1 w2(C) c_2
w2(A) w1(A) r2(B) w1(B) r1(C) w2(C) c_2 c_1
w2(A) w1(A) r2(B) w1(B) r1(C) w2(C) c_1 c_2
w2(A) w1(A) r2(B) w1(B) r1(C) c_1 w2(C) c_2
w2(A) w1(A) w1(B) r1(C) c_1 r2(B) w2(C) c_2
w1(A) w2(A) r2(B) w2(C) c_2 w1(B) r1(C) c_1
w1(A) w2(A) r2(B) w2(C) w1(B) c_2 r1(C) c_1
w1(A) w2(A) r2(B) w1(B) w2(C) c_2 r1(C) c_1
w1(A) w2(A) r2(B) w1(B) r1(C) w2(C) c_2 c_1
w1(A) w2(A) r2(B) w1(B) r1(C) w2(C) c_1 c_2
w1(A) w2(A) r2(B) w1(B) r1(C) c_1 w2(C) c_2

Thus 28 transactions (41 - 13) are both ACR and serailizable.

19.1.5

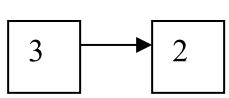
Example of ACR schedule (tx reads values only written by committed trans):
$w_1(A)w_1(B)w_2(A)c_1r_2(B)c_2$

We insert shared and exclusive lock such that the schedule becomes non-strict(releases exclusive lock before commit/abort)
$xl_1(A)w_1(A)ul_1(A)xl_1(B)w_1(B)ul_1(B)xl_2(A)w_2(A)\ ul_2(A)c_1sl_2(B)r_2(B)ul_2(B)c_2$
We see that T_1 release X lock on A ($ul_1(A)$ before c_1.

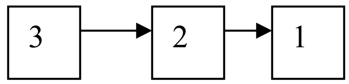
19.2.1a)

After inserting the appropriate lock and unlock requests, the schedule becomes(assuming no delays):
$sl_1(A)r_1(A)sl_2(B)r_2(B)xl_1(C)w_1(C)sl_3(D)r_3(D)sl_4(E)r_4(E)xl_3(B)w_3(B)ul_3(B,D)xl_2(C)w_2(C)ul_2(C,B)xl_4(A)w_4(A)ul_4(A,E)xl_1(D)w_1(D)ul_1(D,A,C)$

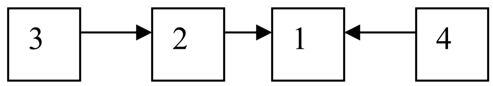
- $sl_1(A)r_1(A)sl_2(B)r_2(B)xl_1(C)w_1(C)\ sl_3(D)r_3(D)sl_4(E)r_4(E)$ proceed fine.
- $xl_3(B)$ is denied since T_2 is holding a lock to B.



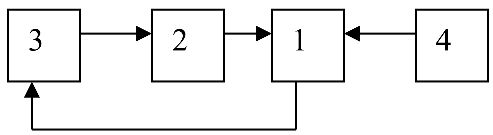
- $xl_2(C)$ is denied since T_1 is holding a lock to C



- $xl_4(A)$ is denied since T_1 is holding a lock to A



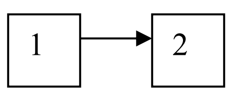
- $xl_1(D)$ is denied since T_3 is holding a lock to D. This causes a deadlock (see the cycle in the waits for graph).



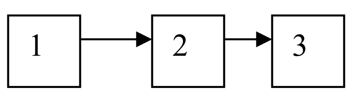
- Assume T_1 is aborted. Releases locks on A and C such that T_2 and T_4 can continue.

<u>19.2.1b)</u>
$sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)xl_1(B)w_1(B)ul_1(B,A)xl_2(C)w_2(C)ul_2(C,B)xl_3(D)w_3(D)ul_3(D,C)$

- $sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)$ proceed fine
- $xl_1(B)$ is denied since T_2 is holding lock to B



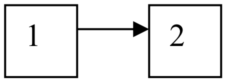
- $xl_2(C)$ is denied since T_3 is holding a lock on C.



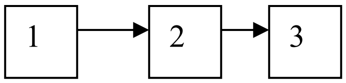
- $xl_3(D)w_3(D)ul_3(D,C)$ complete.
- After T_3 finishes, T_2 completes followed by T_1.

<u>19.2.1c)</u>
$sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)xl_1(B)w_1(B)ul_1(A,B)xl_2(C)w_2(C)ul_2(C,B)xl_3(A)w_3(A)ul_3(A,C)$

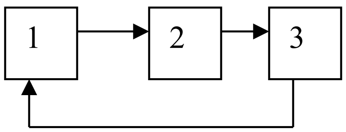
- $sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)$ proceed fine.
- $xl_1(B)$ is denied since T_2 is holding a lock on B



- $xl_2(C)$ is denied since T_3 holds a lock on C



- $xl_3(A)$ is denied since T_1 is holding a lock on A. There is a deadlock as shown by the cycle in the waits-for graph.

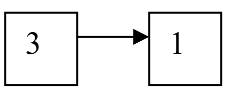


- If T3 is aborted, releases lock on C. T2 can continue. After T2, T1 can continue.

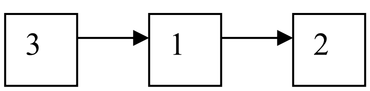
<u>19.2.1d)</u>

$sl_1(A)r_1(A)sl_2(B)r_2(B)xl_1(C)w_1(C)xl_2(D)w_2(D)sl_3(C)r_3(C)ul_3(C)xl_1(B)w_1(B)ul_1(B,A,C)xl_4(D)ul_4(D)xl_2(A)w_2(A)ul_2(A,B,D)$

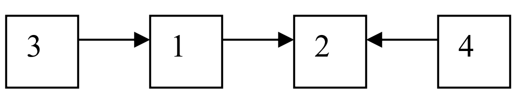
- $sl_1(A)r_1(A)sl_2(B)r_2(B)xl_1(C)w_1(C)xl_2(D)w_2(D)$ proceed fine.
- $sl_3(C)$ is denied since T_1 is holding a lock on C.



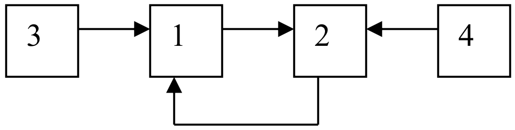
- $xl_1(B)$ is denied since T_2 is holding a lock on B.



- $xl_4(D)$ is denied since T_2 is holding a lock on D.



- $xl_2(A)$ is denied since T_1 is holding a lock on A. This creates a deadlock as shown by the cycle.



- Assume T_2 is aborted. Then T_1 and T_4 can continue. After T_1, T_3 can continue.

19.2.2a)

T_1	**T_2**	**T_3**	**T_4**
$sl_1(A)r_1(A)$			
	$sl_2(B)r_2(B)$		
$xl_1(C)w_1(C)$			
		$sl_3(D)r_3(D)$	
			$sl_4(E)r_4(E)$
		$xl_3(B)$ **Waits** since T_2 is older	
	$xl_2(C)$. **Waits** since T_1 is older.		
			$xl_4(A)$. **Waits** since T_1 is older.
$xl_1(D)$. Aborts T_3 since T_1 is older.		**Wounded.**	

- After T_1 completes, T_2 and T_4 continue. Then T_3 restarts and completes without interference.

19.2.2b)

T1	**T2**	**T3**
$sl_1(A)r_1(A)$		
	$s_l(B)r_2(B)$	
		$sl_3(C)r_3(C)$
$xl_1(B)$. Aborts T_2 since T_1 is older than T_2. Completes.	**Wounded.**	
		$xl_3(D)w_3(D)$. Completes.
	T_2 restarts and completes.	

19.2.2c)

T1	**T2**	**T3**
$sl_1(A)r_1(A)$		
	$sl_2(B)r_2(B)$	
		$sl_3(C)r_3(C)$
$xl_1(B)$. Aborts T_2 since T_1 older than T_2. $w_1(B)$.	**Wounded.**	

Completes.		
		$xl_3(A)w_3(A)$. Completes.
	T_2 restarts and completes.	

19.2.2d)

T1	**T2**	**T3**	**T4**
$sl_1(A)r_1(A)$			
	$sl_2(B)r_2(B)$		
$xl_1(C)w_1(C)$			
	$xl_2(D)w_2(D)$		
		$sl_3(C)r_3(C)$. **Waits** since T1 is older.	
$xl_1(B)$. Aborts T_2 since T_1 is older. $w_1(B)$. Completes.	**Wounded**		
		$sl_3(C)r_3(C)$. Completes.	
			$xl_4(D)$. Completes.
	T_2 restarts and completes.		

19.2.3a)

T1	**T2**	**T3**	**T4**
$sl_1(A)r_1(A)$			
	$sl_2(B)r_2(B)$		
$xl_1(C)w_1(C)$			
		$sl_3(D)r_3(D)$	
			$sl_4(E)r_4(E)$
		$xl_3(B)$. **Dies** since T_2 is older.	
	$xl_2(C)$. **Dies** since T_1 is older.		
			$xl_4(A)$. **Dies** since T_1 is older.
$xl_1(D).w_1(D)$. Completes.			
	T_2 restarts. $sl_2(B)r_2(B)$		
		T_3 restarts. $sl_3(D)r_3(D)$	
			T_4 restarts. $sl_4(E)r_4(E)$
	$xl_2(C)w_2(C)$. Completes		

		$xl_3(B)w_3(B)$. Completes	
			$xl_4(A)w_4(A)$. Completes.

19.2.3b)

T1	**T2**	**T3**
$sl_1(A)r_1(A)$		
	$sl_2(B)r_2(B)$	
		$sl_3(C)r_3(C)$
$xl_1(B)$. Waits since T_1 is older than T_2.		
	$xl_2(C)$. Waits since T_2 is older than T_3.	
		$xl_3(D)w_3(D)$. Completes.
	T_2 resumes and completes.	
T_1 resumes and completes.		

19.2.3c)

T1	**T2**	**T3**
$sl_1(A)r_1(A)$		
	$sl_2(B)r_2(B)$	
		$sl_3(C)r_3(C)$
$xl_1(B)$. **Waits** since T_1 is older than T_2.		
	$xl_2(C)$. **Waits** since T_2 is older than T_3.	
		$xl_3(A)$. **Dies** since T_1 is older.
	T_2 resumes and completes.	
T_1 resumes and completes.		
		T_3 restarts and finishes.

19.2.3d)

T1	**T2**	**T3**	**T4**
$sl_1(A)r_1(A)$			
	$sl_2(B)r_2(B)$		
$xl_1(C)w_1(C)$			
	$xl_2(D)w_2(D)$		
		$sl_3(C)$. **Dies** since T_1 is older.	
$xl_1(B)$. **Waits** since T_1 is older			

than T_2.			
			$xl_4(D)$. **Dies** since T_2 is older.
	$xl_2(A)$. **Dies** since T_1 is older.		
T_1 completes after T_2 dies.			
	T_2 restarts. $sl_2(B)r_2(B)$		
		T_3 restarts. $sl_3(C)r_3(C)$. Completes.	
			T_4 restarts. $xl_4(D)w_4(D)$. Completes.
	$xl_2(D)w_2(D)xl_2(A)w_2(A)$. Completes		

`Given T1, T2, …, Tn, for n > 1,`

19.2.4

There exists a {T1, T2, …, Tn} of waiting transaction such as T1 is waiting for an item held by T2, T2 is waiting for an item held by T3, …, Tn is waiting for an item held by T1. Thus, it is possible to have a wait-for graph with a cycle of length n, for n > 1. However, for n=1, there is no cycle is formed for a node since a transaction doesn't have to wait for a lock held by itself.

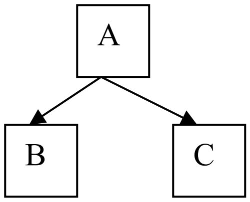
19.2.5

Yes this approach would avoid deadlocks since a transaction would never start until all locks become available. So there would not be a situation where 2 transactions are waiting for each other for locking a resources. Timeouts can be added to prevent starvation.

19.2.6

To construct a waits-for graph for an intention-locking system, the arcs need to be drawn such that:

- the arc goes from transaction waiting for a lock on a node to the transaction holding the lock on the node or its child node that causes the first transaction to wait. For eg., In the tree structure shown, if T1 holds an IX lock on A because it holds a X lock on B (to updated B), and then T2 comes in with a write of A (requiring a X on A), then the arc is drawn from T2 to T1. If multiple transactions hold IX lock on A (for eg, due to write of C by T3), an arc would be drawn from T2 to T3 as well. Similar holds true if T1 and T3 were holding IS locks on A, and T3 came in with a request for X lock on A. Basically the arcs needs to be drawn from the transaction that has to wait for a lock on a node to all the transaction that are holding locks on that node that causes the incoming transaction to waits.



19.2.7)
Suppose T_1 is a transaction that tries to lock elements *A* and *B* in that order. There are also an indefinitely long sequence of transactions T_2, T_3,... that lock *B* and then *A*.

Initially, T_1 locks *A* and T_2 locks *B* and then requests a lock on *A*. T_2 therefore waits for T_1. Meanwhile, T_3 starts and requests a lock on *B*. Thus, T_3 waits for T_2, and the waits-for graph is:

T_3 ----> T_2 ----> T_1

When T_1 requests a lock on *A*, it would cause a cycle, so T_1 is rolled back. Now, T_2 completes and releases its locks. The lock on *A* is given to T_1 and the lock on *B* is given to T_3. Next, T_3 requests a lock on *A*, but has to wait for T_1. Also, T_4 starts up and requests a lock on *B*, thus being forced to wait for T_3. The waits-for graph is now:

T_4 ----> T_3 ----> T_1

The above graph is just like the first, except T_4 and T_3 have replaced T_3 and T_2, respectively. As long as there is a supply of transactions like T_2, T_3, and T_4, they can prevent T_1 from finishing ever.

Notice that the transactions in the example above request locks on *A* and *B* in different orders. Perhaps forcing transactions to request locks in a fixed order will solve the problem. Indeed, if we follow the policy that when a lock becomes available, it is given to a waiting transaction on a first-come-first-served basis, then any transaction that is waiting for a lock will eventually become the transaction that has been waiting for the lock the longest. At that point, the next time the lock becomes available, it will be granted to that transaction, which thus makes some progress. The above observation is the germ of an inductive proof that requesting locks in order plus first-come-first-served will allow every transaction eventually to complete.

However, if the scheduler is able to give locks to any transaction it wishes, then it is easy to make a transaction starve. Suppose T_1 wants a lock on *A* when some other transaction has that lock. As long as there is always another transaction besides T_1 waiting for a lock on *A*, and the scheduler always chooses to give the lock to some transaction other than T_1 when the lock becomes available, T_1 never makes any progress.

Last, let us consider deadlock prevention by timeout. If the duration before a timeout can vary even slightly, then the example given initially for deadlock prevention by avoiding cycles will work. That is, even though T_2 has been waiting slightly longer for a lock than

T_1, if we allow that T_1 might timeout first, then the sequence of events described above could also occur with timeout-based deadlock prevention.

If timeouts occur for any transaction at exactly t seconds from when it first started to wait, then we need a simple modification of the above example. Assume that T_1 is faster at requesting its second lock than any of the other transactions. Then, after T_1 gets a lock on A and T_2 gets a lock on B, T_1 next requests its lock on B and starts to wait. Slightly later, T_2 requests a lock on A. T_3 requests its lock on B just before T_1 times out, so when that timeout occurs, T_2 completes, gives its A-lock to T_1 and its B-lock to T_3. When T_2 next requests a lock on B and starts to wait, this cycle can repeat with T_3 and a new T_4 in place of T_2 and T_3.

<u>19.3.5</u>

The task of the compensating transaction is first to determine whether the file f is still the present one with that name, or whether it has been replaced with a later version.

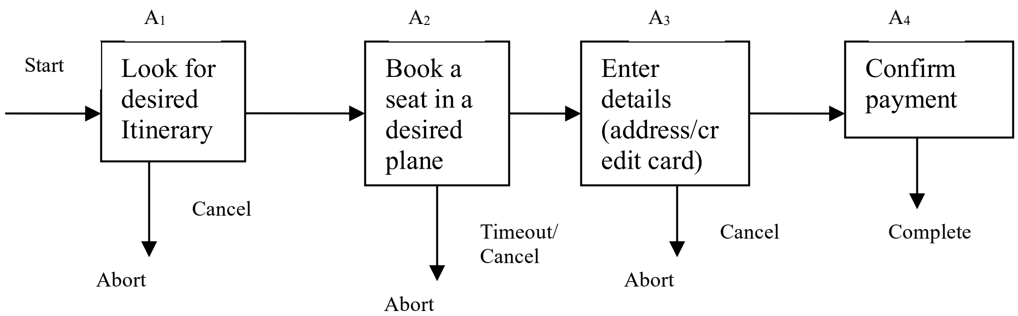
1. If f is still in place, then the compensating transaction must
    1. Restore f' if the latter existed, or
    2. Delete f if not.
2. If f has been overwritten, then there is no need to change the file with that name.

To see that this compensation works, we can consider all the possible cases above, and check that the file system, after compensation, is the same as would be the case had f never been written. If f was overwritten before compensation, then we surely leave the file system in the same state as if f were never written. If f is not overwritten, then we have correctly left the file with that name being either f', if it existed, or left the file system with no file of that name.

There are two interesting issues. First, in order for there to be a compensating transaction, it seems we must both remember the time at which the installation occurred (so we can tell whether the file f has been overwritten when the uninstallation occurs), and we must also remember the file f', at least until such time as the file is overwritten, and the new value is itself guaranteed never to be uninstalled (i.e., the saga that wrote it has finished). These features are not commonly available.

Second, note that file systems do not support serializability. Thus, it is entirely possible that the uninstalled file f has been read, and its contents have influenced other files or the behavior of the system.

<u>19.3.2</u>



List of compensating actions:

- A_1^- : Remove desired itinerary details from database (if saved)
- A_2^-: Put booked seat back in the available pool of seats
- A_3^-: Remove details (address/credit card) from database (if saved)
- A_4^-: Refund charged amount to credit card