Weekly exercises IN3020&4020 (2026)
Week 10: Concurrency Protocols

1 Review

1. Remind yourself what is a transaction, schedule, (conflict-)serialisability, locks (binary, read/write, with conversion), and two-phased locking protocol.

   Solution: See the solutions to the previous week.

2. What do we mean by deadlocks and starvation of a transaction?

   Solution: Deadlock is a situation when we have a cycle in transactions waiting for each other. Starvation occurs if a transaction never acquires all the locks needed for it to complete (i.e., it wait indefinitely) while other transactions in the system continue.

3. Briefly discuss different strategies for avoiding deadlocks in two-phase locking protocols: naive, deadlock prevention (wait-die, wound-wait, no-waiting, cautious-waiting), deadlock detection via wait-for graph, timeout-based

   Solution:

   Naive approaches: conservative two-phase locking, locking to a predefined order of all items

   Deadlock prevention:

   - Wait-die: timestamp-based: If T is older than T' it waits, else it "dies" and restarts with the same timestamp.
   - Wound-wait: timestamp-based If T is older than T', T' is "wounded" and aborts and restarts with the same timestamp.
   - No-waiting: waiting-based: If T tries to lock an item that is locked by T' it aborts and restarts
   - Cautious-waiting: waiting-based: If T' is not waiting for anyone, then T waits, else T aborts and restarts.

Deadlock detection: maintain a wait-for graph (one node for each transaction and an edge whenever a transaction is waiting to lock an item), break a loop by killing a victim if needed

Timeout-based: wait for some (predefined) time, then abort

4. What is a timestamp? Briefly discuss the basic timestamp ordering protocol for concurrency control (without and with the Thomas write rule). How does strict timestamp ordering differ from basic timestamp ordering?

   Solution: A timestamp is an identifier for a transaction. TS are assigned in the order of which the transactions are submitted to the system. Then, if a read or write operation makes a conflict (according to this order), the transaction making the second operation of the conflict is rolled back (in case of write-write conflict, we may ignore the second write instead of rolling back according to the Thomas write rule).

   Strict TS ensures strict schedules. The benefit of this is easy recoverability. Under strict timestamp-ordering protocol, any access (read or write) waits until the previous write is committed.

5. What are the main multiversion protocols for concurrency control and their main ideas?

   Solution: The multiversion protocols for concurrency control maintains multiple versions of the items. The main protocols are timestamp multiversion (each read of an item is of the youngest version older than me, each write creates a new value and is allowed only if no one read a wrong value), two-phase-locking multiversion (two versions: one committed, which is available to everyone except the writing transaction, one local, which is for the writing one), validation-based (3 phases: reading, validation, writing, ordered by the time of validation), snapshot isolation (every transaction works with its own copy of data, writing sets of concurrent transactions must be disjoint).

6. How can you characterise the serial schedules (conflict-)equivalent to the schedules produced by multiversion timestamp-based and validation-based protocols?

   Solution: For a schedule produced by the multiversion timestamp-based protocol the equivalent serial schedule is the one that has the transactions arranged according to the order of the timestamps (i.e., the beginnings of the transactions); in turn, for validation-based protocol the serialised transactions are arranged by the time of validation (i.e., the moment when all the computation is done and only all writings are left).

7. How does the granularity of data items affect the performance of concurrency control? What factors affect selection of granularity size for data items? What is multiple granularity locking? Under what circumstances is it used? What are the main ideas of Multiple-granularity two-phase locking?

   Solution: More coarse grained gives lower degree of concurrency. More fine grained leads to more overhead. If a transaction accesses few records, it is fine to have the lock on record level. If a transaction accesses many record it may be better to have locks on blocks.

   Multiple-granularity two-phase locking:

   (a) Each locking starts in the root and ends in the locked node, using intension locks

   (b) The path to the node is locked with intention locks and the node itself with the usual lock

   (c) Lock-compatibility table must be followed

   (d) Unlocking is symmetric, upgrading is possible

2 Locks and deadlocks, two-phase locking

1. Consider the following two schedules of transactions T_1, T_2 and T_3:

   - $S_1 : r_1(A), r_2(B), r_3(C), r_1(B), r_2(C), r_3(D), w_1(C), w_2(D), w_3(E)$
   - $S_2 : r_1(A), r_2(B), r_3(C), r_1(B), r_2(C), r_3(A), w_1(A), w_2(B), w_3(C)$

   (a) Insert read and write locks, and unlocks into these schedules as follows: place a read lock immediately in front of each read operation that is not followed by a write operation of the same element by the same transaction (i.e., without lock upgrades); place a write lock in front of every other read or write operation; place the necessary unlocks at the end of every transaction.

   (b) Write the schedules of the resulting transactions that try to evaluate the operations in the same order as S_1 and S_2, and violate this order only if necessary (assuming that the transaction manager supports read/write locks).

   Solution:

   (a) $S_1 : rl_1(A), r_1(A), rl_2(B), r_2(B), rl_3(C), r_3(C), rl_1(B), r_1(B), rl_2(C), r_2(C), rl_3(D), r_3(D), wl_1(C), w_1(C), u_1(A), u_1(B), u_1(C), wl_2(D), w_2(D), u_2(B), u_2(C), u_2(D), wl(E), u_2(C), u_3(D), u_3(E)$

T_1	T_2	T_3
$rl_1(A)$		
$r_1(A)$		
	$rl_2(B)$	
	$r_2(B)$	
		$rl_3(C)$
		$r_3(C)$
$rl_1(B)$		
$r_1(B)$		
	$rl_2(C)$	
	$r_2(C)$	
		$rl_3(D)$
		$r_3(D)$
$wl_1(C)$ – wait		
	$wl_2(D)$ – wait	
		$wl_3(E)$
		$w_3(E)$
		$u_3(C)$
		$u_3(D)$
		$u_3(E)$
		c_3
	$w_2(D)$	
	$u_2(B)$	
	$u_2(C)$	
	$u_2(D)$	
	c_2	
$w_1(C)$		
$u_1(A)$		
$u_1(B)$		
$u_1(C)$		
c_1		

Figure 1: Schedule S_1'

$S_2 : wl_1(A), r_1(A), wl_2(B), r_2(B), wl_3(C), r_3(C), rl_1(B), r_1(B),$
$rl_2(C), r_2(C), rl_3(A), r_3(A), w_1(A), u_1(A), u_1(B), w_2(B), u_2(B),$
$u_2(C), w_3(C), u_3(C), u_3(A)$

(b) See Fig. 1 and 2.

2. (*) Explain how to modify the data structures for multiple-mode locks and the algorithms for `read`(X), `write`(X), and `unlock`(X) so that upgrading and downgrading of locks are possible. (Hint: The lock needs to check the transaction ids that hold the lock, if any.)

   Solution: We assume that a list of transaction ids that have read-locked an item is maintained, as well as the (single) transaction id that has write-locked an item. The procedures are shown in Algorithm 1.

3. For each of the (intended) schedules below, assume that read locks

T_1	T_2	T_3
$wl_1(A)$		
$r_1(A)$		
	$wl_2(B)$	
	$r_2(B)$	
		$xl_3(C)$
		$r_3(C)$
$wl_1(B)$ – wait		
	$wl_2(C)$ – wait	
		$wl_3(A)$ – wait
	(deadlock)	

Figure 2: Schedule S_2'

are requested immediately before each read action, and write locks are requested immediately before every write action. Also, unlocks occur immediately after the final action that a trans- action executes. Explain what operations has to wait if a schedule is evaluated under two-phase locking protocol, and whether deadlock occurs. Also explain how the waits-for graph evolves during the execution of the actions. If there are deadlocks, pick a transaction to abort, and show how the schedule continues:

(a) $r_1(A), r_3(B), r_2(C), w_1(B), w_3(C), w_2(D)$

(b) $r_1(A)$,$r_3(B)$,$w_2(C)$,$r_2(D)$,$r_4(E)$,$w_2(B)$,$w_3(C)$,$w_4(A)$,$w_1(D)$,$w_4(A)$

Solution: See Fig. 3 and 4. Observe no deadlock in S_3, but a deadlock between T_2 and T_3 in S_4; observe also that we chosen to abort T_3 in S_4.

4. For the schedule constructed in Question 3(b), explain what happens under the wait-die deadlock prevention protocol, assuming the order of timestamps is the same as the order of subscripts for the transactions—that is, T_1, T_2, T_3, T_4. Assume also that transactions that need to restart do so in the order that they were rolled back.

   Solution: See Fig. 5 (few operations are trimmed at the end); note aborting T_3 and T_4, which cannot wait for older transactions.

5. For the schedule constructed in Question 3(a), explain what happens under the wound-wait deadlock prevention protocol, assuming the order of timestamps is the same as the order of subscripts for the transactions—that is, T_1, T_2, T_3. Assume also that transactions that need to restart do so in the order that they were rolled back.

   Solution: See Fig. 6; note that T_1 wounds T_3 and hence T_3 must abort.

Algorithm 1 Upgrade locking

```
1: procedure READ_LOCK(X, T_n)
2:     if LKD(X) = "unlocked" then
3:         LKD(X) ← "read_locked, List(T_n)"
4:         #READS(X) ← 1
5:     else if LKD(X) = "read_locked, List" then
6:         LKD(X) ← "read_locked, Append(List, T_n)"
7:         #READS(X) ← #READS(X)+1
8:     else if lock(X) = "write_locked, T_n" then
9:         LKD(X) ← "read_locked, List(T_n)" ▷ Belongs to T_n, downgrade lock
10:        #READS(X) ← 1
11:    else
12:        Wait until LKD(X) = "unlocked" and wake up
13:        go to 2
14:
15: procedure WRITE_LOCK(X, T_n)
16:    if LKD(X) = "unlocked" then
17:        LKD(X) ← "write_locked, T_n"
18:    else if LKD(X) = "read_locked" and #READS(X) = 1 and transaction in List = T_n then
19:        LKD(X) = "write_locked, T_n"          ▷ Belongs to T_n, upgrade lock
20:    else
21:        (wait until LKD(X) = "unlocked") or
22:          (LKD(X) = "read_locked, List" and
23:          #READS(X) = 1 and List consists of only T_n)
24:        and wake up
25:        go to 16
26:
27: procedure UNLOCK(X, T_n)
28:    if LKD(X) = "write_locked, T_n" then
29:        LKD(X) ← "unlocked"
30:        Wake one of the waiting transactions if any
31:    else if LKD(X) = "read_locked, List" and T_n ∈ List then
32:        LKD(X) = "read_locked, Remove(List, T_n)
33:        #READS(X) ← #READS(X)−1
34:        if #READS(X) = 0 then
35:            lock(X) = "unlocked"
36:            Wake up one of the waiting transactions if any
```

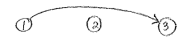
T_1	T_2	T_3	wait-for graph
$rl_1(A)$			
$r_1(A)$			
		$rl_3(B)$	
		$r_3(B)$	
	$rl_2(C)$		
	$r_2(C)$		
$wl_1(B)$ – wait			
			
		$wl_3(C)$ – wait	
	$wl_2(D)$		
	$w_2(D)$		
	$u_2(C)$		
	$u_2(D)$		
	c_2		
			
		$w_3(C)$	
		$u_3(B)$	
		$u_3(C)$	
		c_2	
			
$w_1(B)$			
$u_1(A)$			
$u_1(B)$			
c_1			

Figure 3: Schedule S_3

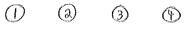
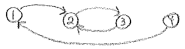
T_1	T_2	T_3	T_4	wait-for graph
$rl_1(A)$				
$r_1(A)$				
		$rl_3(B)$		
		$r_3(B)$		
	$wl_2(C)$			
	$w_2(C)$			
	$rl_2(D)$			
	$r_2(D)$			
			$rl_4(E)$	
			$r_4(E)$	
				
	$wl_2(B)$ – wait			
				
		$wl_3(C)$ – deadlock		
		a_3		
		$u_3(B)$		
		$u_3(C)$		
				
			$wl_4(A)$ – wait	
				
$wl_1(D)$ – wait				
	$wl_2(B)$			
	$w_2(B)$			
	$u_2(C)$			
	$u_2(D)$			
	$u_2(B)$			
	c_2			
$wl_1(D)$				
$w_1(D)$				
$u_1(A)$				
$u_1(D)$				
c_1				
			$wl_4(A)$	
			$w_4(A)$	
			$u_4(E)$	
			$u_4(A)$	
			c_4	

Figure 4: Schedule S_4

T_1	T_2	T_3	T_4	wait-for graph
$rl_1(A)$				
$r_1(A)$				
		$rl_3(B)$		
		$r_3(B)$		
	$wl_2(C)$			
	$w_2(C)$			
	$rl_2(D)$			
	$r_2(D)$			
			$rl_4(E)$	
			$r_4(E)$	
	$wl_2(B)$ – wait			
		$wl_3(C)$ – die		
		a_3		
		$u_3(B)$		
		$u_3(C)$		
	$w_2(B)$			
	$u_2(C)$			
	$u_2(D)$			
	$u_2(B)$			
	c_2			
			$wl_4(A)$ – die	
			a_4	
			$u_4(E)$	
$wl_1(D)$				
$w_1(D)$				
$u_1(A)$				
$u_1(D)$				
c_1				
		(restart)		
			(restart)	

Figure 5: Schedule S_4'

T_1	T_2	T_3
$rl_1(A)$		
$r_1(A)$		
		$rl_3(B)$
		$r_3(B)$
	$rl_2(C)$	
	$r_2(C)$	
$wl_1(B)$ – wound T_3		
		a_3
		$u_3(B)$
$wl_1(B)$		
$w_1(B)$		
$u_1(A)$		
$u_1(B)$		
c_1		
	$wl_2(D)$	
	$w_2(D)$	
	$u_2(C)$	
	$u_2(D)$	
	c_2	
		(restart)

Figure 6: Schedule S_3'

3 Timestamps

1. Consider the following schedules (which include the beginning operations):

   (a) $b_1, r_1(A), b_2, w_2(B), r_2(A), w_1(B)$

   (b) $b_1, b_2, r_1(A), r_2(B), w_2(A), w_1(B)$

   Show what happens as each executes under the timestamp-based concurrency control protocol.

   Solution:

   (a) In this case, T_1 aborts since $TS(T_1) < WriteTS(B)$:

T_1	T_2
b_1	
$r_1(A)$	
	b_2
	$w_2(B)$
	$r_2(A)$
	c_2
$w_1(B)$	
a_1	

   (b) In this case, T_1 aborts since $TS(T_1) < ReadTS(B)$

T_1	T_2
b_1	
	b_2
$r_1(A)$	
	$r_2(B)$
	$w_2(A)$
	c_2
$w_1(B)$	
a_1	

2. Give an example of an application of Thomas write rule (and explain its purpose).

   Solution: The purpose of the Thomas writing rules is to improve both basic and strict timestamp protocols in case of write-write conflict. This is done by ignoring write operations that have become obsolete. Example:

T_1	T_2
$r_1(X)$	
	$w_2(X)$
	c_2
~~$w_2(X)$~~	
c_2	

T_1's write is ignored because it is outdated

4 Multiversioning and multigranularity concurrency

1. Consider the following schedules:

   (a) $S_1 : b_1, r_1(A); b_2, r_2(A); r_1(B); w_2(A); w_1(A); r_2(B); w_2(B);$

   (b) $S_2 : b_1, r_1(A); b_2, r_2(B); w_2(B); r_1(B); w_1(A); w_1(B);$

   (c) $S_3 : b_1, r_1(A); b_2, r_2(A); b_3, r_3(A); w_1(B); r_3(B); w_2(A); w_3(A); w_2(B);$

   Describe what happens when these schedules evaluate under the multiversioning timestamp-based protocol.

   Solution:

   (a) Note that T_1 is aborted because its write to A should have been read by T_2:

T_1	T_2	Comments
b_1		
$r_1(A)$		ReadTS(A_0) = TS(T_0)
	b_2	
	$r_2(A)$	ReadTS(A_0) = TS(T_1)
$r_1(B)$		ReadTS(B_0) = TS(T_0)
	$w_2(A)$	create A_1 with Read/WriteTS(A_1)=TS(T_2)
$w_1(A)$ – rejected		because ReadTS(A_0) > TS(T_0)
a_1		T_1 aborts
	$r_2(B)$	ReadTS(B_0) = TS(T_1)
	$w_2(B)$	create B_1: Read/WriteTS(B_1)=TS(T_2)
	c_2	

   (b) Note that T_1 is aborted because its write to B should have been read by T_2:

T_1	T_2	Comments
b_1		
$r_1(A)$		ReadTS(A_0) = TS(T_0)
	b_2	
	$r_2(B)$	ReadTS(B_0) = TS(T_1)
	$w_2(B)$	create B_1 Read/WriteTS(B_1) = TS(T_2)
	c_2	
$r_1(B)$		T_1 reads B_0
$w_1(A)$		create A_1 Read/WriteTS(A_1) = TS(T_1)
$w_1(B)$ – rejected		because ReadTS(B_0) > TS(T_0)
a_1		T_1 must abort

(c) Here T_2 must abort, but since it has no writes it is not necessary to do a cascade abort:

T_1	T_2	T_3	Comments
b_1			
$r_1(A)$			ReadTS(A_0) = TS(T_1)
	b_2		
	$r_2(A)$		ReadTS(A_0) = TS(T_2)
		b_3	
		$r_3(A)$	ReadTS(A_0) = TS(T_3)
$w_1(B)$			create B_1 Read/WriteTS(B_1) = TS(T_1)
c_1			
		$r_3(B)$	ReadTS(B_1) = TS(T_3)
	$w_2(A)$ – rejected		because ReadTS(A_0) > TS(T_2)
	a_2		T_2 must abort
		$w_3(A)$	create A_1 Read/WriteTS(A_1) = TS(T_3)
		c_3	

2. Do the same as in Question 1, but using multiversion two-phase locking.

Solution:

(a) Deadlock because both transactions are waiting for locks:

T_1	T_2	Comments
b_1		
$rl_1(A)$		reads committed
$r_1(A)$		
	b_2	
	$rl_2(A)$	reads committed
	$r_2(A)$	
$rl_1(B)$		
$r_1(B)$		
	$wl_2(A)$	
	$w_2(A)$	writes to local copy
$wl_1(A)$		– wait for T_2
	$rl_2(B)$	
	$r_2(B)$	
	$wl_2(B)$	
	$w_2(B)$	
	$cl_2(A)$	– wait for T_1 deadlock

(b) Here, T_1 is aborted since its B write should have been read by T_2:

T_1	T_2	Comments
b_1		
$rl_1(A)$		
$r_1(A)$		reading committed A
	b_2	
	$rl_2(B)$	
	$r_2(B)$	reading committed B
	$wl_2(B)$	
	$w_2(B)$	writing local B
	$cl_2(B)$	
	$u_2(B)$	
	c_2	
$rl_1(B)$		
$r_1(B)$		reading committed B
$wl_1(A)$		
$w_1(A)$		writing local A
$wl_1(B)$		
$w_1(B)$		writing local B
$cl_1(A)$		
$cl_1(B)$		
$u_1(A)$		
$u_1(B)$		
c_1		

(c) Here T_1 commits, but there is a deadlock between T_2 and T_3:

T_1	T_2	T_3	Comments
b_1			
$rl_1(A)$			
$r_1(A)$			
	b_2		
	$rl_2(A)$		
	$r_2(A)$		
		b_3	
		$rl_3(A)$	
		$r_3(A)$	
$wl_1(B)$			
$w_1(B)$			to local B
$cl_1(B)$			
$u_1(A)$			
$u_1(B)$			
c_1			
		$rl_3(B)$	
		$r_3(B)$	
	$wl_2(A)$		
	$w_2(A)$		to local A
		$wl_3(A)$	lock rejected
			T_3 waiting for T_2
	$wl_2(B)$		
	$w_2(B)$		
	$cl_2(A)$		lock rejected
			T_2 waiting for T_3
			deadlock

3. Same as above, but with snapshot isolation.