Weekly exercises IN3020 and IN4020
Week 11

May 24, 2022

1 Review

1. What do we mean by starvation of a transaction?

   Solution: 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 normally.

2. Discuss different strategies for avoiding deadlocks in two-phase locking protocols

   Solution:

   - 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.

3. What is a wait-for graph?

   Solution: A wait-for graph is used to detect deadlocks. One node for each transaction and an edge whenever a transaction is waiting to lock an item.

4. What is a timestamp? Discuss the timestamp ordering protocol for concurrency control. 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. Strict TS ensures strict schedules. The benefit of this is easy recoverability. Strict timestamp-ordering waits until the previous write is committed.

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

   Solution: The multiversion protocols for concurrency control maintains multiple versions of the items. The main protocols are timestamp multiversion and two-phase-lock multiversion. Snapshot isolation can also be mentioned. The advantage is that we can have more concurrency when read operations can read an older version to maintain serialisability, the disadvantage is that we use more memory.

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

   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 leve. 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

   (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

8. Open-ended discussion questions based on Section 21.6 of the book: What are the specific properties of B+-trees that do not allow to efficiently use two-phase locking and other usual concurrency control protocols for B+-tree indexes? Can you think of approaches to concurrency control for B+-tree indexes?

   Solution: While it is possible to use two-phase locking on B+-trees, holding a lock on an index page until the shrinking phase can cause an undue amount of transaction blocking because searching always starts at the root. While it would not be a problem to simultaneously update different subbranches of the tree.

2 Locks and deadlocks, two-phase locking

1. For each of the 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 shared and exclusive locks, and insert unlock operations. Place a shared lock immediately in front of each read action that is not followed by a write action of the same element by the same transaction (i.e., without lock upgrades). Place an exclusive lock in front of every other read or write action. Place the necessary unlocks at the end of every transaction.

   (b) Tell what happens when each schedule is run by a scheduler that supports shared and exclusive locks.

   (c) Insert shared and exclusive locks in a way that allows upgrading. Place a shared lock in front of every read, an exclusive lock in front of every write, and place the necessary unlocks at the ends of the transactions.

   (d) Tell what happens when each schedule from previous item is run by a scheduler that supports shared locks, exclusive locks, and upgrading.

   Solution:

   (a)
   - $S_1 : sl_1(A), r_1(A), sl_2(B), r_2(B), sl_3(C), r_3(C), sl_1(B), r_1(B), sl_2(C), r_2(C), sl_3(D), r_3(D), xl_1(C), w_(C), u_1(A), u_1(B), u_1(C), xl_2(D), w_2(D), u_2(B), u_2(C), u_2(D), xl(E), u_2(C), u_3(D), u_3(E)$
   - $S_2 : xl_1(A), r_1(A), xl_2(B), r_2(B), xl_3(C), r_3(C), sl_1(B), r_1(B), sl_2(C), r_2(C), sl_3(A), r_3(A), w_1(A), u_1(A), u_1(B), w_2(B), u_2(B), u_2(C), w_3(C), u_3(C), u_3(A)$

(b)

T_1	T_2	T_3
$sl_1(A)$		
$r_1(A)$		
	$sl_2(B)$	
	$r_2(B)$	
		$sl_3(C)$
		$r_3(C)$
$sl_1(B)$		
$r_1(B)$		
	$sl_2(C)$	
	$r_2(C)$	
		$sl_3(D)$
		$r_3(D)$
$xl_1(C)$ -reject		
	$xl_2(D)$ -reject	
		$xl_3(E)$
		$w_3(E)$
		$u_3(C)$
		$u_3(D)$
		$u_3(E)$
		c_3
	$xl_2(D)$	
	$w_2(D)$	
	$u_2(B)$	
	$u_2(C)$	
	$u_2(D)$	
	c_2	
$xl_1(C)$		
$w_1(C)$		
$u_1(A)$		
$u_1(B)$		
$u_1(C)$		
c_1		

T_1	T_2	T_3
$xl_1(A)$		
$r_1(A)$		
	$xl_2(B)$	
	$r_2(B)$	
		$xl_3(C)$
		$r_3(C)$
$sl_1(B)$ - reject		
	$sl_2(C)$ - reject	
		$sl_3(A)$ - reject

(c)
- For S_1 it makes no difference to use upgrade locks.
- $S_2 : sl_1(A), r_1(A), sl_2(B), r_2(B), sl_3(C), r_3(C), sl_1(B), r_1(B), sl_2(C), r_2(C), sl_3(A), r_3(A), xl_1(A), w_1(A), u_1(A), u_1(B), xl_2(B), w_2(B), u_2(B), u_2(C), xl_3(C), w_3(C), u_3(C), u_3(A)$

2. Explain how to modify the data structures for multiple-mode locks and the algorithms for `read\_lock(X)`, `write\_lock(X)`, and `unlock(X)` so that upgrading and downgrading of locks are possible. (Hint: The lock needs to check the transaction $id(s)$ 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 on the following page.

3. For each of the schedules below, assume that shared locks are requested immediately before each read action, and exclusive locks are requested immediately before every write action. Also, unlocks occur immediately after the final action that a trans- action executes. Tell what actions are denied, and whether deadlock occurs. Also tell 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 sequence of actions continues.

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

   (b) $r_1(A), r_3(B), w_2(C), r_2(D), r_4(E), w_2(B), w_3(C), w_4(A), w_1(D), w_4(A)$[1]

   Solution:

[1] updated 7/4

Algorithm 1 Upgrade locking

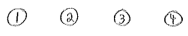
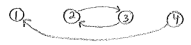
```
1: procedure READ_LOCK(X, T_n)
2:     if lock(X) = "unlocked" then
3:         lock(X) ← "read_locked, List(T_n)"
4:         no_of_reads(X) ← 1
5:     else if lock(X) = "read_locked, List" then
6:         lock(X) ← "read_locked, Append(List,T_n)"
7:         no_of_reads(X) ← no_of_reads(X) + 1
8: 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

(a)

T_1	T_2	T_3	wait-for graph
$sl_1(A)$			
$r_1(A)$			
		$sl_3(B)$	
		$r_3(B)$	
	$sl_2(C)$		
	$r_2(C)$		
$xl_1(B)$			
–rejected			
			
		$xl_3(C)$	
		–rejected	
	$xl_2(D)$		
	$w_2(D)$		
	$u_2(C)$		
	$u_2(D)$		
	c_2		
			
		$xl_3(C)$	
		$w_3(C)$	
		$u_3(B)$	
		$u_3(C)$	
		c_2	
			
$xl_1(B)$			
$w_1(B)$			
$u_1(A)$			
$u_1(B)$			
c_1			

No deadlock in S_1.

(b)

T_1	T_2	T_3	T_4	wait-for graph
$sl_1(A)$				
$r_1(A)$				
		$sl_3(B)$		
		$r_3(B)$		
	$xl_2(C)$			
	$w_2(C)$			
	$sl_2(D)$			
	$r_2(D)$			
			$sl_4(E)$	
			$r_4(E)$	
				
	$xl_2(B)$			
	–rejected			
				
		$xl_3(C)$		
		–rejected[2]		
				
			$xl_4(A)$	
			–rejected	
				
$xl_1(D)$				
–rejected[3]				
		a_3[4]		
		$u_3(B)$		
		$u_3(C)$		
	$xl_2(B)$			
	$w_2(B)$			
	$u_2(C)$			
	$u_2(D)$			
	$u_2(B)$			
	c_2			
$xl_1(D)$				
$w_1(D)$				
$u_1(A)$				
$u_1(D)$				
c_1				
			$xl_4(A)$	
			$w_4(A)$	
			$u_4(E)$	
			$u_4(A)$	
			c_4	

4. For each of the schedules below (the same as in the previous question), tell what happens under the wait-die deadlock avoidance system. Assume the order of deadlock-timestamps is the same as the order of subscripts for the transactions, that is, T_1 , T_2 , T_3 , and T_4. Also assume that transactions that need to restart do so in the order that

[1] deadlock between T_2 and T_3

[2] T_1 and T_4 are also involved in the deadlock

[3] We choose to abort T_3

they were rolled back.

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

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

Solution:

(a) Still no deadlock for S_1.

(b)

T_1	T_2	T_3	T_4	wait-for graph
$sl_1(A)$ $r_1(A)$				
		$sl_3(B)$ $r_3(B)$		
	$xl_2(C)$ $w_2(C)$ $sl_2(D)$ $r_2(D)$			
			$sl_4(E)$ $r_4(E)$	
	$xl_2(B)$ –rejected			
		$xl_3(C)$ a_3[5] $u_3(B)$ $u_3(C)$		
	$xl_2(B)$ $w_2(B)$ $u_2(C)$ $u_2(D)$ $u_2(B)$ c_2			
			$xl_4(A)$ a_4[6] $u_4(E)$	
$xl_1(D)$ $w_1(D)$ $u_1(A)$ $u_1(D)$ c_1				

5. For each of the schedules in the previous question, explain what happens under the wound-wait deadlock avoidance system.

Solution:

[4] Aborting T_3, cannot wait for older transaction

[5] Aborting T_4, cannot wait for older transaction

(a)

T_1	T_2	T_3
$sl_1(A)$		
$r_1(A)$		
		$sl_3(B)$
		$r_3(B)$
	$sl_2(C)$	
	$r_2(C)$	
$xl_1(B)$[7]		
		a_3
		$u_3(B)$
$xl_1(B)$		
$w_1(B)$		
$u_1(A)$		
$u_1(B)$		
c_1		
	$xl_2(D)$	
	$w_2(D)$	
	$u_2(C)$	
	$u_2(D)$	
	c_2	

[6] T_1 wounds T_3. T_3 must abort.

(b)

T_1	T_2	T_3	T_4
$sl_1(A)$			
$r_1(A)$			
		$sl_3(B)$	
		$r_3(B)$	
	$xl_2(C)$		
	$w_2(C)$		
	$sl_2(D)$		
	$r_2(D)$		
			$sl_4(E)$
			$r_4(E)$
	$xl_2(B)$[8]		
		a_3	
		$u_3(B)$	
	$xl_2(B)$		
	$w_2(B)$		
	$u_2(C)$		
	$u_2(D)$		
	$u_2(B)$		
	c_2		
			$xl_4(A)$[9]
			–rejected
$xl_1(D)$			
$w_1(D)$			
$u_1(A)$			
$u_1(D)$			
c_1			
			$xl_4(A)$
			$w_4(A)$
			$u_4(A)$
			$u_4(E)$
			$u_4(A)$

3 Timestamps

1. Consider the following schedules, which include the beginning operations start events. Tell what happens as each executes under the timestamp-based concurrency protocol.

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

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

   Solution:

[7] T_2 wounds T_3, T_3 must abort.
[8] T_4 (newer) can wait

(a)

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

T_1 aborts since $TS(T_1) < write_TS(B)$

(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	

T_1 aborts since $TS(T_1) < read_TS(B)$

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 multi-versioning timestamp-based protocol.

Solution:

(a)

T_1	T_2	Comments
b_1		
$r_1(A)$		$read_TS(A_0) = TS(T_1)$
	b_2	
	$r_2(A)$	$read_TS(A_0) = TS(T_2)$
$r_1(B)$		$read_TS(B_0) = TS(T_1)$
	$w_2(A)$	create A_1: $read/write_TS(A_1) = TS(T_2)$
~~$w_1(A)$~~		$read_TS(A_0) > TS(T_1)$ write rejected
a_1		T_1 aborts
	$r_2(B)$	$read_TS(B_0) = TS(T_2)$
	$w_2(B)$	create B_1: $read/write_TS(B_1) = TS(T_2)$
	c_2	

T_1 is aborted because its write to A should have been read by T_2.

(b)

T_1	T_2	Comments
b_1		
$r_1(A)$		$read_TS(A_0) = TS(T_1)$
	b_2	
	$r_2(B)$	$read_TS(B_0) = TS(T_2)$
	$w_2(B)$	create B_1 $read/write_TS(B_1) = TS(T_2)$
	c_2	
$r_1(B)$		T_1 reads B_0
$w_1(A)$		create A_1 $read/write_TS(A_1) = TS(T_1)$
~~$w_1(B)$~~		$read_TS(B_0) > TS(T_1)$ write rejected
a_1		T_1 must abort

T_1 is aborted because its write to B should have been read by T_2.

(c)

T_1	T_2	T_3	Comments
b_1			
$r_1(A)$			$read_TS(A_0) = TS(T_1)$
	b_2		
	$r_2(A)$		$read_TS(A_0) = TS(T_2)$
		b_3	
		$r_3(A)$	$read_TS(A_0) = TS(T_3)$
$w_1(B)$			create B_1 $read/write_TS(B_1) = TS(T_1)$
c_1			
		$r_3(B)$	$read_TS(B_1) = TS(T_3)$
	~~$w_2(A)$~~		$read_TS(A_0) > TS(T_2)$ write rejected
	a_2		T_2 must abort
		$w_3(A)$	create A_1 $read/write_TS(A_1) = TS(T_3)$
		c_3	

Here T_2 must abort, but since it has no writes it is not necessary to do a cascade abort.

2. Same as above, but using multiversion two-phase locking

   Solution:

(a)

T_1	T_2	Comments
b_1		
$read_lock_1(A)$		reads committed
$r_1(A)$		
	b_2	
	$read_lock_2(A)$	reads committed
	$r_2(A)$	
$read_lock_1(B)$		
$r_1(B)$		
	$write_lock_2(A)$	
	$w_2(A)$	writes to local copy
$write_lock_1(A)$		–wait for T_2
	$read_lock_2(B)$	
	$r_2(B)$	
	$write_lock_2(B)$	
	$w_2(B)$	
	$certify_lock_2(A)$	–wait for T_1 deadlock

Deadlock because both transactions are waiting for locks.

(b)

T_1	T_2	Comments
b_1		
$read_lock_1(A)$		
$r_1(A)$		reading committed A
	b_2	
	$read_lock_2(B)$	
	$r_2(B)$	reading committed B
	$write_lock_2(B)$	
	$w_2(B)$	writing local B
	$certify_lock_2(B)$	
	$u_2(B)$	
	c_2	
$read_lock_1(B)$		
$r_1(B)$		reading committed B
$write_lock_1(A)$		
$w_1(A)$		writing local A
$write_lock_1(B)$		
$w_1(B)$		writing local B
$certify_lock_1(A)$		
$certify_lock_1(B)$		
$u_1(A)$		
$u_1(B)$		
c_1		

T_1 is aborted because its write to B should have been read by T_2.

(c)

T_1	T_2	T_3	Comments
b_1			
$read_lock_1(A)$			
$r_1(A)$			
	b_2		
	$read_lock_2(A)$		
	$r_2(A)$		
		b_3	
		$read_lock_3(A)$	
		$r_3(A)$	
$write_lock_1(B)$			
$w_1(B)$			to local B
$certify_lock_1(B)$			
$u_1(A)$			
$u_1(B)$			
c_1			
		$read_lock_3(B)$	
		$r_3(B)$	
	$write_lock_2(A)$		
	$w_2(A)$		to local A
		$write_lock_3(A)$	lock rejected
			T_3 waiting for T_2
	$write_lock_2(B)$		
	$w_2(B)$		
	$certify_lock_2(A)$		lock rejected
			T_2 waiting for T_3
			deadlock

T_1 commits, but deadlock between T_2 and T_3.

3. Give an example of a schedule that has to roll-back when evaluated under validation-based protocol.

   Solution: $S_1 : r_1(X), w_1(X), r_2(X), w_2(X), w_1(Y)$
   When T_2 starts its writing phase, there will be an non-empty intersection between $WS(T_1)$ and $WS(T_2) \cup RS(T_2)$.