COMP207 Lab Exercises
Tutorial 4 (Week 6)

The exercises below provide the opportunity to practice the concepts and methods discussed during previous week's lectures (lectures 10–12). If you haven't done so, it is worthwhile to spend some time on making yourself familiar with these concepts and methods. Don't worry if you cannot solve all the exercises during the lab session, but try to tackle at least one or two of them. If at some point you do not know how to proceed, you could review the relevant material from the lecture notes and return to the exercise later.

Solutions will be provided after the last lab session of this week on Thursday.

Timestamp-based deadlock detection

As pointed out in Lecture 10, deadlocks may arise even if transactions are scheduled using strict two-phase locking (strict 2PL). Lectures 10 and 11 covered several techniques for deadlock detection, among them two techniques based on timestamps: *wait-die* and *wound-wait*. The following example illustrates the two methods:

Example 1 (wait-die)**.** Consider the following two strict 2PL transactions from Lecture 10:

Transaction T_1	Transaction T_2
lock(X)	lock(Y)
read_item(X)	read_item(Y)
$X := X + 100$	$Y := Y + 100$
write_item(X)	write_item(Y)
lock(Y)	lock(X)
read_item(Y)	read_item(X)
$Y := Y + 100$	$X := X + 100$
write_item(Y)	write_item(X)
commit	commit
unlock(X)	unlock(X)
unlock(Y)	unlock(Y)

As pointed out in Lecture 10, we might end up in a deadlock if we execute the operations of the two transactions in the "wrong" way. For example, after executing lines 1–4 of T_1 and then executing lines 1–4 of T_2 we are in the situation that neither T_1 nor T_2 can proceed, because T_1 waits for T_2 to unlock Y and T_2 waits for T_1 to unlock X.

Let us see how the *wait-die* scheme deals with the deadlock. Assume transaction T_1 arrives earlier than transaction T_2. Then, T_1 has a lower timestamp than T_2:

$$\text{TS}(T_1) < \text{TS}(T_2).$$

Figure 1 shows the schedule according to which T_1 and T_2 are executed before the deadlock arises. At time step 9, there are two possibilities: the transaction manager requests to execute

Time	T_1	T_2	**Comment**
0			
1	`lock`(X)		Lock request granted
2	`read_item`(X)		
3	$X := X + 100$		
4	`write_item`(X)		
5		`lock`(Y)	Lock request granted
6		`read_item`(Y)	
7		$Y := Y + 100$	
8		`write_item`(Y)	

Figure 1: The schedule that corresponds to executing the first four operations of transaction T_1 from Example 1, followed by the first four operations of transaction T_2.

the next operation of T_1 (`lock`(Y)) or it requests to execute the next operation of T_2 (`lock`(X)).

Let us first consider the case that it requests to execute the next operation of T_1 (`lock`(Y)). Since T_2 holds a lock on Y, the scheduler denies the lock request. In other words, T_1 waits for T_2 to unlock Y. Since T_1 is older than T_2 ($\texttt{TS}(T_1) < \texttt{TS}(T_2)$), the scheduler does not abort T_1 – it allows T_1 to wait further.

Let us now consider the case that the transaction manager requests to execute the next operation of T_2 (`lock`(X)). Since T_1 holds a lock on X, the scheduler denies the lock request. In other words, T_2 waits for T_1 to unlock X. Since T_2 is younger than T_1 ($\texttt{TS}(T_2) > \texttt{TS}(T_1)$), the scheduler aborts T_2 and restarts it with the same timestamp. When the scheduler aborts T_2, it also releases the lock it held on Y, so the deadlock is resolved and T_1 can continue its execution. Note that the wait-die scheme prevents T_2 from starting while T_1 still holds a lock on Y (if it attempts to lock Y it would be aborted, because T_1 now holds a lock on Y and T_2 is younger than T_1). Hence, T_2 only starts after T_1 has finished.

A summary of all the events can be found in Figure 2.

Example 2 (wound-wait)**.** Let us now see how the wound-wait scheme deals with the deadlock described in Example 1. Again, we assume that transaction T_1 arrives earlier than transaction T_2, so T_1 has a lower timestamp than T_2:

$$\texttt{TS}(T_1) < \texttt{TS}(T_2).$$

Figure 1 shows the schedule according to which T_1 and T_2 are executed before the deadlock arises. At time step 9, we have the same two possibilities as before: the transaction manager requests to execute the next operation of T_1 (`lock`(Y)) or it requests to execute the next operation of T_2 (`lock`(X)).

Let us first consider the case that it requests to execute the next operation of T_1 (`lock`(Y)). Since T_2 holds a lock on Y, the scheduler denies the lock request. In other words, T_1 waits for T_2 to unlock Y. Since T_1 is older than T_2 ($\texttt{TS}(T_1) < \texttt{TS}(T_2)$), it “wounds” T_2, which means that the scheduler aborts T_2 and restarts it with the same timestamp. When the scheduler aborts T_2, it also releases the lock it held on Y, so the deadlock is resolved and T_1 can continue its execution. As under the wait-die scheme, the scheduler here prevents T_2 from executing its

Time	**T_1**	**T_2**	**Comment**
1	`lock`(X)		Lock request granted
2	`read_item`(X)		
3	$X := X + 100$		
4	`write_item`(X)		
5		`lock`(Y)	Lock request granted
6		`read_item`(Y)	
7		$Y := Y + 100$	
8		`write_item`(Y)	
9		*abort, unlock Y, restart*	T_2 waits for T_1 to unlock X. Since T_2 is younger than T_1, we abort T_2, release its lock on Y, and restart it with the same timestamp (wait-die scheme).
10	`lock`(Y)		Lock request granted
11	`read_item`(Y)		
12	$Y := Y + 100$		
13	`write_item`(Y)		
14	`commit`		
15	`unlock`(X)		
16	`unlock`(Y)		
17		`lock`(Y)	Lock request granted
18		`read_item`(Y)	
19		$Y := Y + 100$	
20		`write_item`(Y)	
21		`lock`(X)	Lock request granted
22		`read_item`(X)	
23		$X := X + 100$	
24		`write_item`(X)	
25		`commit`	
26		`unlock`(X)	
27		`unlock`(Y)	

Figure 2: Schedule for T_1 and T_2 enforced by wait-die scheme. T_1 is assumed to have started earlier. We also assume that the first four operations of T_1 were executed first, followed by the first four operations of T_2.

first operation while T_1 still holds a lock on Y, however in a slightly different way: instead of aborting T_2 once it tries to execute its first operation, T_2 is now allowed to wait until T_1 unlocks Y (since T_2 is younger than T_1). T_2 executes after T_1 has finished.

Let us now consider the case that the transaction manager requests to execute the next operation of T_2 (`lock`(X)). Since T_1 holds a lock on X, the scheduler denies the lock request. In other words, T_2 waits for T_1 to unlock X. Since T_2 is younger than T_1 ($\mathtt{TS}(T_2) > \mathtt{TS}(T_1)$), the scheduler allows T_2 to wait further.

A summary of all the events can be found in Figure 3. Note that Figure 3 is essentially the same as Figure 2, except for the reason for aborting T_2 (time step 9).

Exercise 1. Consider the following schedules:

- S_1: $xl_1(X)$; $r_1(X)$; $sl_2(Y)$; $r_2(Y)$; $xl_2(X)$; $w_2(X)$; $u_2(X)$; $u_2(Y)$; $w_1(X)$; $u_1(X)$
- S_2: $sl_1(X)$; $r_1(X)$; $xl_2(Y)$; $r_2(Y)$; $xl_1(Y)$; $r_1(Y)$; $w_2(Y)$; $u_2(Y)$; $w_1(Y)$; $u_1(X)$; $u_1(Y)$

For each of the schedules, decide if a lock request is denied, and if so give the first lock request that is denied and say what happens in this case under the

(a) wait-die scheme;

(b) wound-wait scheme.

Assume that T_1 arrives earlier than T_2.

Time	**T_1**	**T_2**	**Comment**
1	`lock`(X)		Lock request granted
2	`read_item`(X)		
3	$X := X + 100$		
4	`write_item`(X)		
5		`lock`(Y)	Lock request granted
6		`read_item`(Y)	
7		$Y := Y + 100$	
8		`write_item`(Y)	
9		*abort, unlock Y, restart*	T_1 waits for T_2 to unlock Y. Since T_1 is older than T_2, we abort T_2, release its lock on Y, and restart it with the same timestamp (wound-wait scheme).
10	`lock`(Y)		Lock request granted
11	`read_item`(Y)		
12	$Y := Y + 100$		
13	`write_item`(Y)		
14	`commit`		
15	`unlock`(X)		
16	`unlock`(Y)		
17		`lock`(Y)	Lock request granted
18		`read_item`(Y)	
19		$Y := Y + 100$	
20		`write_item`(Y)	
21		`lock`(X)	Lock request granted
22		`read_item`(X)	
23		$X := X + 100$	
24		`write_item`(X)	
25		`commit`	
26		`unlock`(X)	
27		`unlock`(Y)	

Figure 3: Schedule for T_1 and T_2 enforced by wound-wait scheme. T_1 is assumed to have started earlier. We also assume that the first four operations of T_1 were executed first, followed by the first four operations of T_2.

Timestamp-based scheduling

Exercise 2 (Exercise 18.8.1 in [1])**.** Below are several sequences of start events and read/write operations (here, st_i means that transaction T_i starts):

(a) $st_1;\ st_2;\ r_1(X);\ r_2(Y);\ w_2(X);\ w_1(Y)$

(b) $st_1;\ r_1(X);\ st_2;\ w_2(Y);\ r_2(X);\ w_1(Y)$

(c) $st_1;\ st_2;\ st_3;\ r_1(X);\ r_2(Y);\ w_1(Z);\ r_3(Y);\ r_3(Z);\ w_2(Y);\ w_3(X)$

(d) $st_1;\ st_3;\ st_2;\ r_1(X);\ r_2(Y);\ w_1(Z);\ r_3(Y);\ r_3(Z);\ w_2(Y);\ w_3(X)$

Tell what happens as each of the sequences executes under a timestamp-based scheduler. Assume that the read and write times of all items are 0 at the beginning of the sequence.

References

[1] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book*. Pearson Education, 2nd edition, 2009.