CSCE-310 Database Systems
Homework 9
Raymond Zhu
923008555

Problem 1:

a) A := A+B and B := A+B

Actions	t	s	Mem A	Mem B	Disk A	Disk B
READ(A,t)	5		5		5	10
READ(B,s)	5	10	5	10	5	10
t := t+s	15	10	5	10	5	10
WRITE(A,t)	15	10	15	10	5	10
READ(A,t)	5	10	15	10	5	10
s := t+s	5	15	15	10	5	10
WRITE(B,s)	5	15	15	15	5	10
OUTPUT(A)	5	15	15	15	15	10
OUTPUT(B)	5	15	15	15	15	15

It is possible to assure consistency if a crash were to occur as long as all the steps are executed properly where the information is stored on both disk A and B. If the database were to crash before OUTPUT(A), the disks are still consistent. Only if the database were to crash upon OUTPUT(B), the database would be inconsistent since A and B are not the same.

c) A := B+1 and B := A+1

Actions	t	s	Mem A	Mem B	Disk A	Disk B
READ(A,t)	5		5		5	10
READ(B,s)	5	10	5	10	5	10
t := t+1	6	10	5	10	5	10
WRITE(B,t)	6	10	5	6	5	10
s := s+1	6	11	5	6	5	10
WRITE(A,s)	6	11	11	6	5	10
OUTPUT(A)	6	11	11	6	11	10

OUTPUT(B)	6	11	11	6	11	6

It is possible to assure consistency if a crash were to occur as long as all the steps are executed properly where the information is stored on both disk A and B. If the database were to crash before OUTPUT(A), the disks are still consistent. Only if the database were to crash upon OUTPUT(B), the database would be inconsistent since A and B are not the same.

Problem 2:

a) < START U>;
   If the program were to crash with the last logged message as < START U >, the recovery manager will look at the previously logged messages and notice that both T and U have not yet been committed. Because such a case has not yet occurred, the recovery manager can easily rollback the value of A where A was originally set to 10 by transaction T.

c) < T, E, 30 >;
   If the program were to crash with the last logged message as <T, E, 30>, the recovery manager will look at the previously logged messages and notice that U has already been committed. Because U has already been committed changes to those values are already in disk, so the recovery manager can only roll back changes made by transaction T. Looking at the logs, the recovery manager would restore the values A, C and E back to 10, 30 and 50 respectively.

Problem 3:

a) < START U >;
   If the program were to crash with the last logged message as < START U >, the recovery manager will notice that both T and U transactions have not yet been committed. Because such transactions have not yet been committed and there's a redo log, no action needs to be taken since the database is still in a consistent state.

c) < T, E, 30 >;
   If the program were to crash with the last logged message as <T, E, 30>, the recovery manager will notice that transaction U has already been committed. Given such a case, changes made by T will be treated as if they never happened, however, the recovery manager will need to do something about U to restore the database back to a consistent state. The redo logging will have the manager look back at the logged data and revert values B and D changed by transaction U back to 20 and 40 respectively.

Problem 4:

a) < START U >;
   If the program were to crash with the last logged message as < START U >, the recovery manager will notice that both transaction U and T have not yet been committed. Because T modified the variable A, the manager will rollback A to the old value of 10. Since U did not make any changes, nothing has to be done.

c) < T, E, 50, 51 >;

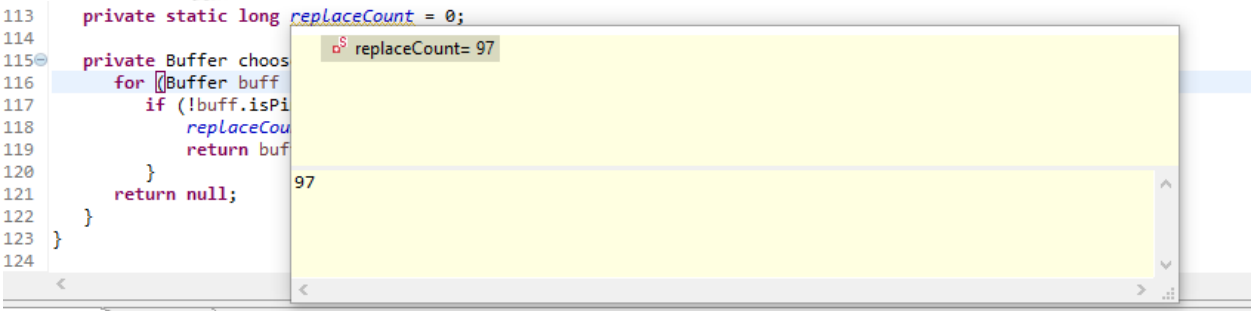
If the program were to crash with the last logged message as < T, E, 30, 51 >, the recovery manager will notice that transaction U has already been committed and transaction T has modified a couple of variables. And so, the recovery manager will rollback transaction T back to the old values while setting changes made by transaction U to its new values.

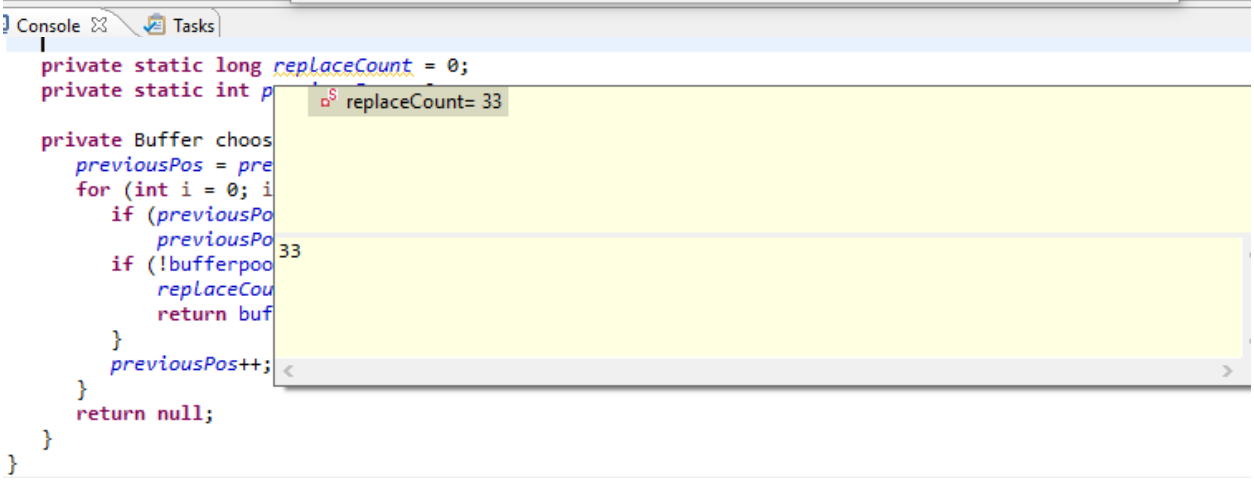
Problem 5:

- Changes made to chooseUnpinnedBuffer() include adding a new field, private static int previousPos = 0;, to keep track of the position after returning. The for loop was changed to be able to easily get the index of the buffer in the bufferpool. It is clear that there's a difference when using a "clock" algorithm compared to the naive implementation.

```
private static long replaceCount = 0;
private static int previousPos = 0;

private Buffer chooseUnpinnedBuffer() {
   previousPos = previousPos + 1;
   for (int i = 0; i < bufferpool.length; ++i) {
      if (previousPos >= available()) //RZ
          previousPos = 0;
      if (!bufferpool[previousPos].isPinned()) {
          replaceCount++;
          return bufferpool[previousPos];
      }
      previousPos++; // RZ
   }
   return null;
}
```





Problem 6:

a) A = 5 and B = 10

T1	T2	A	B
READ(A,t)		5	
t := t+2		7	
WRITE(A,t)		7	
READ(B,t)			10
t := t*3			30
WRITE(B,t)			30
	READ(B,s)		30
	s := s*2		60
	WRITE(B,s)		**60**
	READ(A,s)	7	
	s := s+3	10	
	WRITE(A,s)	**10**	

T1	T2	A	B
	READ(B,s)		10
	s := s*2		20
	WRITE(B,s)		20
	READ(A,s)	5	
	s := s+3	8	
	WRITE(A,s)	8	
READ(A,t)		8	
t := t+2		10	
WRITE(A,t)		**10**	
READ(B,t)			20

t := t*3			60
WRITE(B,t)			**60**

As it can be seen, both A and B end in the same state with the value 10 and 60 respectively either when scheduling with (T1, T2) or (T2, T1).

b) As defined in the textbook, a serializable schedule is serializable if there is a serial schedule such that for every initial database state, the effects of both schedules are the same. In such a case, it can be seen that both tables above in part (a) are serializable given that both effects lead to the same state. In the case of a non-serializable schedule, the 12 actions above cannot make a non-serializable schedule. It's impossible to create an inconsistent outcome given the 12 actions because the changes made to A and B are of different operations. Unless there were other operations, interleaving the tasks could result in an inconsistent state. However, for these 12 tasks it's not possible.

Problem 7:

c) $w_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_3(C)$; $r_3(C)$;
There are three transactions and from the first one we can see that transaction 3 will have initially locked onto element A. After writing to A transaction 3 will release the element. Afterwards, after waiting on transaction 3, transaction 1 will lock both element A and B. After reading A the transaction will then write the value into B and release both elements. Next, transaction 2 will lock both element B and C, reading B and writing to C, and release until the transaction is complete. Lastly, transaction 3 will return and lock C, read C, and release the element.

d) $r_1(A)$; $r_2(A)$; $w_1(B)$; $w_2(B)$; $r_1(B)$; $r_2(B)$; $w_2(C)$; $w_1(D)$;
There are two transactions and from the first one we can see that transaction 1 will have initially locked onto element A and will release the element after reading it. Afterwards, after waiting on transaction 1, transaction 2 will lock onto element A and release the element after reading it. Transaction 1 is then granted lock on to B, at the same time transaction 2 attempts to access element B but is denied. And so, transaction 2 will need to wait on transaction 1 to modify element B. After both transactions have finished, transaction 1 then locks onto element B, reads B, and then releases the lock on element B. Transaction 2 then locks both elements B and C, reading B and writing to C, and then releasing both elements. Lastly, transaction 1 locks element D. writes to D, and then releases D.

Problem 8:

c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(C)$; $w_2(D)$; $w_3(E)$;
   I. **$sl_1(A)$;** $r_1(A)$; **$sl_2(B)$;** $r_2(B)$; **$sl_3(C)$;** $r_3(C)$; **$sl_1(B)$;** $r_1(B)$; **$sl_2(C)$;** $r_2(C)$; **$sl_3(D)$;** $r_3(D)$; $w_1(C)$; **$u_1(A)$; $u_1(B)$;** $w_2(D)$; **$u_2(B)$; $u_2(C)$;** $w_3(E)$; **$u_3(C)$; $u_3(D)$;**

II. When the schedule runs with the listed locks above the scheduler will initially deny transaction 1 and 2 from writing to element C and D respectively because C is locked by transaction 2 and D is locked by transaction 3. The schedule will only complete once all elements have been completely unlocked.

d) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(A)$; $w_2(B)$ $w_3(C)$;

I. **$xl_1(A)$;** $r_1(A)$; **$xl_2(B)$;** $r_2(B)$; **$xl_3(C)$;** $r_3(C)$; **$sl_1(B)$;** $r_1(B)$; **$sl_2(C)$;** $r_2(C)$; **$sl_3(D)$;** $r_3(D)$; $w_1(A)$; **$u_1(A)$; $u_1(B)$;** $w_2(B)$; **$u_2(B)$; $u_2(C)$;** $w_3(C)$; **$u_3(C)$; $u_3(D)$;**

II. When the schedule runs with the listed locks above the scheduler will initially deny transaction 1 and 2 from accessing the shared lock B and C respectively. This is because transaction 3 has been granted share lock on C and transaction 2 has been granted share lock on B. The schedule will complete once all elements have been completely unlocked.

Problem 9:

$r_1(A)$; $r_1(B)$; $w_1(A)$; $w_1(B)$;

a) Two-phase locked, and strict

$r_1(A)$; $r_1(B)$; $w_1(A)$; $w_1(B)$;
$r_1(A)$; $r_1(B)$; $w_1(B)$; $w_1(A)$;

$r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
$r_1(B)$; $w_1(B)$; $r_1(A)$; $w_1(A)$;

$r_1(B)$; $r_1(A)$; $w_1(A)$; $w_1(B)$;
$r_1(B)$; $r_1(A)$; $w_1(B)$; $w_1(A)$;

There are six different outcomes for a single type

b) Two-phase locked, but not strict

$r_1(A)$; $w_1(A)$; $u_1(A)$; $r_1(B)$; $w_1(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $u_1(A)$; $w_1(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$; $u_1(A)$;

$r_1(A)$; $r_1(B)$; $w_1(A)$; $w_1(B)$; $u_1(A)$;
$r_1(A)$; $r_1(B)$; $w_1(B)$; $w_1(A)$; $u_1(A)$;
$r_1(A)$; $r_1(B)$; $w_1(A)$; $u_1(A)$; $w_1(B)$;

$r_1(B)$; $r_1(A)$; $w_1(B)$; $w_1(A)$; $u_1(A)$;
$r_1(B)$; $r_1(A)$; $w_1(A)$; $w_1(B)$; $u_1(A)$;
$r_1(B)$; $r_1(A)$; $w_1(A)$; $u_1(A)$; $w_1(B)$;

There are nine different outcomes for a single type

Problem 10:

b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;

As defined in the textbook, a deadlock is where several transactions are waiting for a resource held by one of the other transactions and no progress can be done. Specifically, a wait-die deadlock compares the time a transaction has to wait for the resource. If the wait time is longer than the elapsed time of current transaction occupying the resource, the waiting transaction dies.

In our above example, no cycle can be formed between the three transactions and therefore a deadlock cannot be formed. Thus, the wait-die avoidance system does not apply for this situation.

c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;

From the example above a deadlock is formed because there's a cycle. Transaction 3 is dependent on the resource held by transaction 1. Looking at the order of the schedule, transaction 1 is waiting to write element B and won't unlock element A until B has been released by transaction 2. Unfortunately, this won't happen anytime soon because transaction 2 is dependent on transaction 3 to release resource C until it can write to C and release B.

As it can be seen, because transaction 3 is waiting for resource A, the wait time for transaction 3 will exceed the elapsed time of A. And so, transaction 3 will eventually die and release resource C for transaction 2, and so forth, until both A and B are released by transaction 1.