CSE 5331 | Project 1

Phase 2 - Simulate a rigorous two phase locking protocol (2PL) with wound wait method for deadlock prevention and concurrency control

UPDATE

1. Implemented `wait-die` method for deadlock prevention for the **extra credit**

Contributors

Student Id	Student Name	Contribution
1001767678	Harshavardhan Ramamurthy	class-structure, write_lock(), end_transaction(), wound_wait(), commit(), abort()
1001767676	Karan Rajpal	main-driver, read_lock(), begin_transaction(), execute_operation(), unlock(), wait_die()

Instructions

This program uses only python's internal libraries and does not need any other dependencies

1. All input files are placed in `inputs/` directory in the project folder with the naming convention `input[0-9].txt`
2. If you are using a linux based system, then do steps 3 and 4. If not, then skip to step 5.
3. Provide execute permissions to the `simulate.sh` file by running `chmod +x simulate.sh` in your console
4. Run `./simulate.sh` in your console. This will
    - Simulate both `wound-wait` and `wait-die` on all the input files present in the `inputs/` directory
    - Display output on the console
    - Save the outputs to `outputs/wound-wait/` and `outputs/wait-die/` respectively for reference

   Demo:

```
❯ ./simulate.sh
Input file: inputs/input1.txt, Output also saved to: outputs/wound-wait/output1.txt
Using wound-wait for deadlock prevention

 1 - Transaction T1 started - b1
 2 - T1 applied read-lock on item Y - r1(Y)
 3 - T1 upgraded the lock on item Y to write
 ...
```

```
 16 - Transaction T2 committed. Releasing all locks held - e2
     T2 released lock on item Y
 ...
 ...
 ...
 ...
 Input file: inputs/input1.txt, Output also saved to: outputs/wait-
die/output1.txt
 Using wait-die for deadlock prevention

 1 - Transaction T1 started - b1
 2 - T1 applied read-lock on item Y - r1(Y)
 3 - T1 upgraded the lock on item Y to write
 4 - T1 applied read-lock on item Z - r1(Z)
 5 - Transaction T2 started - b2
 ...

# For reference after execution
❯ tree outputs
outputs
 ├── wait-die
 │   ├── output1.txt
 │   ├── output2.txt
 │   ├── output3.txt
 │   └── output4.txt
 └── wound-wait
     ├── output1.txt
     ├── output2.txt
     ├── output3.txt
     └── output4.txt

```

5. Execute the program using `python main.py <prevention-method> inputs/<input_file>.txt`

   Example 1: `python main.py wound-wait inputs/input1.txt`

   And your output should look like:

```
❯ python main.py wound-wait inputs/input1.txt

Using wound-wait for deadlock prevention

1 - Transaction T1 started - b1
2 - T1 applied read-lock on item Y - r1(Y)
3 - T1 upgraded the lock on item Y to write
4 - T1 applied read-lock on item Z - r1(Z)
5 - Transaction T2 started - b2
6 - Item Y already write-locked by T1. Using wound-wait to resolve
conflict - r2(Y)
7 - T2 blocked for read-lock on item Y. REASON: Older transaction T1
has applied write lock on it. - r2(Y)
8 - Transaction T3 started - b3
9 - T3 applied read-lock on item Z - r3(Z)
10 - Older transaction T1 applied write-lock on Z - w1(Z)
11 - Aborting transaction T3. REASON: Older transaction T1 applied
write-lock on Z - w1(Z)
    T3 released lock on item Z
```

```
12 - T1 upgraded the lock on item Z to write
13 - Transaction T1 committed. Releasing all locks held - e1
    T1 released lock on item Y
    T1 released lock on item Z
14 - T2 resumed operation from wait-list for item Y. - e1
15 - T2 applied read-lock on item Y - r2(Y)
16 - Transaction T2 committed. Releasing all locks held - e2
    T2 released lock on item Y
```

Example 2: `python main.py wait-die inputs/input1.txt`
And your output should look like:

```
❯ python main.py wait-die inputs/input1.txt
Using wait-die for deadlock prevention

 1 - Transaction T1 started - b1
 2 - T1 applied read-lock on item Y - r1(Y)
 3 - T1 upgraded the lock on item Y to write
 4 - T1 applied read-lock on item Z - r1(Z)
 5 - Transaction T2 started - b2
 6 - Item Y already write-locked by T1. Using wait-die to resolve conflict - r2(Y)
 7 - T1 aborted since a younger transaction T2 applied write-lock on item Y - r2(Y)
 8 - Aborting transaction T1. REASON: T1 aborted since a younger transaction T2 applied write-lock on item Y - r2(Y)
     T1 released lock on item Y
     T1 released lock on item Z
 9 - T2 applied write-lock on item Y - r2(Y)
 10 - Transaction T3 started - b3
 11 - T3 applied read-lock on item Z - r3(Z)
 12 - T3 upgraded the lock on item Z to write
 13 - Transaction T3 committed. Releasing all locks held - e3
     T3 released lock on item Z
 14 - Transaction T2 committed. Releasing all locks held - e2
```

Pseudo Code

main-driver

Reads statements from input file and drives the program

```
timestamp := 0 # To track the transaction's timestamp
READ input_file
FOR each line in input_file:
    current_transaction := contents_of_transaction_table
    IF current_transaction is blocked THEN
        ADD operation to list of waiting operations for current_transaction in transaction table
    ELSE
        IF current_transaction is aborted THEN
            Disregard operation
        ELSE
            execute_operation()
```

begin_transaction(transaction_number, transation_timestamp)

Starts a transaction by adding an new entry in the transaction table with status as 'active'

```
def begin_transaction(transaction_id, transation_timestamp):
    transaction_id := timestamp + transaction_number
    INSERT (tid:transaction_id,timestamp: transaction_timestamp, status:
'active', items:{}) for this transaction in the transaction_table
```

execute_operation(operation)

Executes operation sent as an argument

```
def execute_operation(operation):
    TOKENIZE line into operation and item

    IF operation = 'b' THEN
        INCREMENT timestamp
        begin_transaction()
    IF operation = 'r' THEN
        Get item to be read from the variable/data structure storing it
        Apply read_lock on the variable using readlock(variable)
    IF operation = 'w' THEN
        APPLY write lock using writelock()
    IF operation = 'e' THEN
        COMMIT transaction using commit()
```

read_lock()

Retrieves all records present in the lock_table for an item. If no records are present, then records are inserted to lock_table with appropriate status and the transaction table is updated as well. If the item is locked by a non-conflicting transaction, then it is unlocked. If the transcation is write-locked by another transcation then wound_wait() is executed.

> Implemented as a block of code.

```
def read_lock():
    record := entry for the item in transaction_table
    IF record for item NOT IN lock_table THEN
        INSERT item into the lock_table with 'read' as state of lock
        DISPLAY the reansaction that has readlocked the item
    ELSE
        IF item is writelocked THEN
            # Conflict in transaction
            # Use wound wait to make a resolution
            wound_wait()
        ELSE
            UPDATE item in lock_table
            APPEND tid to transaction_holding
            UPDATE items OF transaction IN transaction_table
            DISPLAY the transaction that has readlocked the item
```

write_lock()

Retrieves all records present in the lock_table for an item. If item is read-locked by the same transaction, then the lock status is updated to a write-lock. If the item is unlocked beforehand, then the status is updated to write-locked directly. If the item is locked by another transaction, then wound_wait() is executed.

> Implemented as a block of code.

```
def write_lock():
    item := record for the item from lock_table
    IF item is already locked THEN
        GET type of lock from transaction that has currently locked the item
    IF item locked by same transaction THEN
        UPDATE lock table entry for the item from readlock to writelock
        DISPLAY the transaction that has upgraded the lock
    ELSE
        IF item IS NOT locked THEN
            UPDATE status OF item TO writelocked
            APPEND tid to the transaction_holding
            DISPLAY the transaction that has held the lock
        ELSE
            IF item locked by another transaction THEN
                call wound_wait() to resolve conflict
            ELSE
                INSERT entry to lock_table
                DISPLAY the transaction that has writelocked the item
```

commit()

Unlocks all present locks for a transaction and updates the status to committed in the transaction table.

```
def commit():
    items := {items_locked_by_transaction}
    FOR EACH item in items:
        unlock(item) # to unlock items
    UPDATE status OF transaction in transaction_table to "committed"
    DISPLAY that the transaction has been committed
```

abort()

Unlocks all present locks for a transaction and updates the status to aborted in the transaction table.

```
def abort():
    items := {items_locked_by_transaction}
    FOR EACH item in items:
        unlock(item) # to unlock items
    UPDATE status OF transaction in transaction_table to "aborted"
```

unlock()

Unlocks item in the transaction table by updating the status accordingly. If any transactions are waiting for this item the the lock is granted to them.

```
def unlock():
```

```
    FOR EACH transaction in lock_table
        IF transaction is waiting THEN
            resumed_transaction := {transaction}
            DISPLAY resumed_transaction has resumed operation
            GET tid of resumed_transaction from lock_table
            REMOVE tid of resumed_transaction from transaction_waiting in lock_table
            APPEND tid of resumed_transaction to transaction_holding in lock_table
            UPDATE status of resumed_transaction in transaction_table to "active"
            # Execute waiting operations from transaction table
            operations_list := {operations from transaction_table}
            FOR EACH operation in operations_list
                execute_operation()
        ELSE
            REMOVE tid of transaction from transaction_holding in lock_table
            UPDATE state of transaction in lock_table to "unlocked"
            REMOVE tid of transaction from items in transaction_table
```

woundwait()

Used to decide which transaction will wait and which will abort when a deadlock occurs based on the timestamp stored in the transaction table.

```
def wound_wait():
    request_timestamp := timestamp_of_requesting_transaction
    hold_timestamp := timestamp_holding_transaction_lock
    IF request_timestamp < hold_timestamp THEN
        DISPLAY requesting_transaction will abort
        abort(requesting_transaction) # Will be restarted later with same timestamp
    ELSE
        # requesting_transaction will wait
        APPEND tid of requesting_transaction to transaction_waiting in lock_table
        UPDATE status of requesting_transaction in transaction_table to "blocked"
        DISPLAY requesting_transaction is blocked
```

wait_die()

Uses the `wait-die` approach to resolve conflict when a transaction is trying to lock an item that is already locked.

```
def wait_die():
    request_timestamp := timestamp_of_requesting_transaction
    hold_timestamp := timestamp_holding_transaction_lock
    IF request_timestamp < hold_timestamp THEN
        # requesting_transaction will wait
        APPEND tid of requesting_transaction to transaction_waiting in lock_table
        UPDATE status of requesting_transaction in transaction_table to "blocked"
        DISPLAY requesting_transaction is blocked
    ELSE
        DISPLAY requestion_transaction will abort
        abort(requesting_transaction)
        # Will be restarted later with same timestamp

```

Data Structures Proposed

Transaction Table

Implemented as a class with the following members

Attribute	Description	Data type
tid	Transaction ID	`int`
timestamp	Transaction Timestamp	`int`
items	Items the current transaction holds	`list`
status	State of current transaction(active/committed/blocked/aborted)	`string`
operations	Operations in the waiting transaction	`list`

Lock Table

Implemented as a class with the following members

Attribute	Description	Data type
item	The item locked or unlocked by the transaction	`int`
tid_holding	List of transactions currently holding the item	`list`
tid_waiting	List of transactions currently waiting to hold the item	`list`
state	Current state of the item (r/w)	`str`

Record

Class to simplify accessing of the various parts of the input line such as Transaction, Operation and Item

Attribute	Description	Data type
tid	Transaction ID	`int`
item	The item the transaction is attempting apply lock on	`str`
operation	Operation being performed - (b/e/r/w)	`str`