**Project 1**

Intermediate Report- Rigorous 2 Phase Locking protocol with wound-wait method for dealing with deadlock.

**Team:** Sneha Srinivas(1001667832)

Pratik Barhate(1001649826)

**Programming Language:** Python

**Data Structures:**

The following are the table structures that will be used to capture the transaction and locking information:

1. Transaction Table:- transaction\_table{}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Transaction\_ID | Transaction\_State | Timestamp | List of Waiting Operations | Transaction Items | Phase |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

1. Locking Table:- lock\_table{}

|  |  |  |  |
| --- | --- | --- | --- |
| Date Item Name | Lock Status | Locking Transactions | List of Transactions waiting |
|  |  |  |  |
|  |  |  |  |

The implementation of the simulation will make use of Array List of Python in a Dictionary for storing and processing the transactions and the locks applied on the data items.

transaction\_table={}

lock\_table={}

**Implementation:**

**Step 1**:- Read from file

The program will read the schedule text file line by line and will recognize the operation, transaction\_id, item\_id, status.

This is done by using the line.split(“ ”) to capture the elements of the operations as states above.

Pseudo Code:

*Each line read is stored as a String* ***line****.*

First item in the array of line[0] is assigned to **operation**

Split with delimiter as “ “ and store it in **line\_transaction**

The second item in the array of the line\_transaction[0][1] is assigned to **transaction\_id**

status\_lock = read\_lock

current\_transaction = Current Transaction id is extracted from the transaction table

A method: read\_item(line) is implemented in the Step 2.

**Step 2**:- Identify the type of operation obtained from step 1 and preform the respective action.

Pseudo Code:

status\_RL = read\_lock

current\_transaction = Current Transaction id is extracted from the transaction table

read\_item(line):

if the phase of current\_transaction == TRUE

(when it is in growing phase(Wherein read, write , upgrade is done, and the data items can only acquire the locks, and none can be released or downgraded))

if there is entry of item id in the lock table

Instantiating the current lock to the corresponding item id in the lock table

if the current lock is read\_lock

if the transaction\_id is the first transaction or not. If not append it to the table.

else if it is write\_lock

check if there are multiple read locks on the item

(Before upgrading an item lock status from read lock to write lock we must)

current\_lock['locking\_transactions'].append(transaction\_id)

current\_lock['lock\_status'] = status\_RL

current\_transaction ['trans\_item'].append(item\_id) else if the transaction\_id is not in

current\_lock[‘locking\_transaction’]

check for the priority of conflicting transaction based on wound wait protocol

else if the item is not available in the lock table, then make new entry in the lock table

lock\_table[item\_id] = {'lock\_status': status\_RL, 'locking\_transactions': [transaction\_id], 'list\_of\_transactions\_waiting': []}

transaction\_table[transaction\_id]['trans\_item'].append(item\_id)

status\_WL = write\_lock

current\_transaction = Current Transaction id is extracted from the transaction table

write\_item(line):

if the phase of transaction\_table[transaction\_id] == TRUE

(when it is in growing phase(Wherein read, write , upgrade is done, and the data items can only acquire the locks, and none can be released or downgraded))

if there is entry of item id in the lock table

Instantiating the current lock to the corresponding item id in the lock table

if the current lock is read\_lock

Only one transaction can hold this item with read lock in order to directly upgrade to the write lock.

current\_lock['lock\_status'] = ‘write\_lock’

else if:

(the list has more than one transaction holding the read lock on current item check which transaction should be executed first)

check if there are multiple read locks on the item

(Before upgrading an item lock status from read lock to write lock we must)

for t in t\_list:

(check for the priority of conflicting transaction based on wound wait protocol)

protocol = woundwait(t1,transaction\_id,operation,item\_id,line)

(check the timestamp of the current transaction against all the transactions in the list to abort or terminate one among them using wait die protocol)

if protocol == "aborted":

break

else if current\_lock['lock\_status'] = ‘write\_lock’

(check if the transaction\_id exists in the t\_list)

if transaction\_id not in t\_list:

t = t\_list[0]

woundwait(t,transaction\_id,operation,item\_id,line)

else if the item is not available in the lock table, then make new entry in the lock table

lock\_table[item\_id] = {'lock\_status': status\_WL, 'locking\_transactions': [transaction\_id], 'list\_of\_transactions\_waiting': []}

transaction\_table[transaction\_id]['trans\_item'].append(item\_id)

**Step 3**:- Apply wound-wait method for dealing with deadlock when there exists a lock for an operation on a data-item and another lock is required.

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

if TS(Ti) < TS(Tj) − that is Ti, which is requesting a conflicting lock, is older than Tj

then Tj is wounded(aborted).

else if TS(Ti) > TS(Tj) − that is Ti is younger than Tj

then Ti waits(gets blocked).

This scheme allows the younger transaction to wait but kills the older one.

The wound-wait function applied to this rigorous 2PL protocol is as below-

Pseudo Code:

woundwait(t1, t2, operation, item\_id, line)

if the Trans1 == Trans2

return

if either of the Trans2 is either not blocked or not aborted

(compare the Timestamps of both the transaction)

if TS2<TS1

(then wound Trans1 by aborting it)

call end\_transaction(t1,t\_aborted,line) on Trans1

else :

Block Trans2 and dd it to the list\_of\_waiting\_operations

current\_lock is the item\_ID from the lock\_table{}

t\_list is the ‘locking\_transactions’ in the lock\_table under the item\_ID

assigning “aborted” to t\_aborted

assigning “blocked” to t\_blocked

assigning the transaction\_table[t1] and transaction\_table[t2] to Trans1 and Trans2 respectively

extracting and assigning the Trans1 timestamp to TS1. Similarly, with TS2.

**Step 5:-** end\_transaction

When current transaction encountered should release all the existing locks placed on all the data items.

Furthermore, a check should be done to find out the transactions that were blocked due to the data items being locked by this transaction. Thus, those transactions should *try* re-lock the data items such that there is no deadlock in the process.

Pseudo Code:

end\_transaction(t,t\_state,line)

read\_lock is assigned to status\_RL

write\_lock is assigned to status\_WL

active is assigned to t\_active

aborted is assigned to t\_aborted

committed is assigned to t\_committed

Phase of the current transaction is set to false since it goes to shrinking phase

(Wherein unlock and downgrading is done)

iterating d\_item in the current\_transaction items:

current\_lock, t\_list, waiting\_list is all assigned to the corresponding values in the lock table

the transaction holding the item must be removed from locking transactions in lock table(shrinking phase)

if there are no transaction waiting on the item, then it proceeds to the next operation on the line

continue

First transaction waiting in the queue is popped in order to put the transaction back to the active state

Corresponding operation for the waiting transaction is being extracted

if the waiting operation is a read operation

if it has conflicting write lock

then Downgrading the lock status to read lock

current\_lock['locking\_transactions'] = [first\_waiting\_trans]

Update the transaction id's state to active in the transaction table

nxt\_op = The next waiting operation is being popped

if nxt\_op['operation'] == "e":

end\_transaction(first\_waiting\_trans,t\_committed,line)

(end transaction is recursively called since next operation is end\_transaction)

iterating t through the waiting\_t\_list:

Extracts the row against transaction id

Get the first waiting operation

if t2['operation'] == "r":

popping out the first waiting operation

remove transaction from the list of waiting transaction

The popped transaction is set to active again

if nxt\_op['operation'] == "e":

Previous transaction is committed, and the next operation will be in the queue

end\_transaction(first\_waiting\_trans, t\_committed, line

(end transaction is recursively called since next operation is end\_transaction)

else if the waiting operation is a write operation

if it has conflicting read lock

if t\_list length is 0 or 1 && t\_list[0] == first\_waiting\_trans

(since we are upgrading the current item lock from the read lock to write lock it must be either 0 or 1 transaction holding it)

write\_item is appended with the first\_waiting\_trans

current\_lock['lock\_status'] = status\_WL

Get the first waiting transaction

Update the transaction id's state to active in the transaction table

nxt\_op2 = pops the next waiting operation

if nxt\_op2['operation'] == "e":

Previous transaction is committed, and the next operation will be in the queue

end\_transaction(first\_waiting\_trans, t\_committed, line)

(end transaction is recursively called since next operation is end\_transaction)

else if the list has more locking transactions then the next possible

iterate t1 over t\_list:

protocols = call woundwait(t1,first\_waiting\_trans, op1['operation'],op1['item\_id'],line)

if protocols == t\_aborted: break

else current\_lock['lock\_status'] == status\_WL

t\_list = [first\_waiting\_trans]

Updating the transaction id's state to active in the transaction table

nxt\_op2 = pops the next waiting operation

if nxt\_op2['operation'] == "e"

end\_transaction(first\_waiting\_trans, t\_committed, line)

(end transaction is recursively called since next operation is end\_transaction)

Since the current transaction list varies for each transaction id it is cleared on each iteration