Concurrency Control in Transactional Systems CS5280 PROJECT (MID-SEM REVIEW)

GROUP 5

Indian Institute of Technology, Hyderabad

Introduction

Project Motivation

Our aim is to develop a **practical O2PL scheduler algorithm**.

- Transactions acquire conflicting locks under *ordered shared relationships*, replacing strict mutual exclusion with **execution-order dependencies**. This ensures conflict serializabilty via precedence constraints.
- The order of operation execution follows the order of lock acquisition, decoupling mutual exclusion duration from critical section execution.
- Although the non-blocking nature of ordered sharing of locks causes commit-phase delays, in order to enforce dependency order, these delays do not block other transactions from executing read and write operations.

CORE CONCEPT: ORDERED SHARING

- **OS1: Lock Acquisition Order $\rightarrow$ Execution Order** If transaction t_j acquires lock $ql_j(x)$ after t_i's lock $pl_i(x)$:

$$pl_i(x) <_s ql_j(x) \Rightarrow p_i(x) <_s q_j(x)$$

- **OS2: Lock Release Rule** The lock release is delayed for order-dependent transactions. If t_j is order dependent on t_i, then t_j enters **on-hold** state and cannot release any locks.

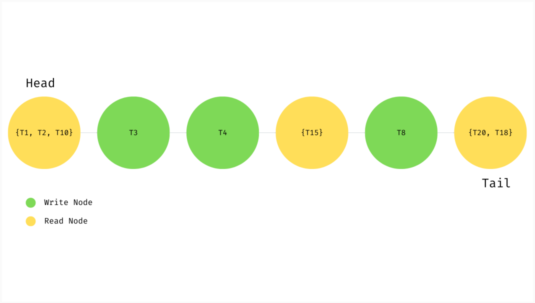
Note

Without loss of generality, the scheduler releases all locks at the end of the schedule for all transactions.

High level Idea

- We require the conflicting operations to be executed in the order of their arrival, if it is valid.
- The serializability of the transactions is guaranteed by the virtue of Waits for graph. I.e., whenever serializability of the operations fail, it causes the waits for graph to have a cycle, which lead to abort.
- We are planning to use a linked list queue for each data item as shown in 4

List



Notice that the read node has a set of transactions, whereas the write node only has a transaction. All compatible operations will be grouped in the set. And whenever a conflicting operation is encountered, a new node is added to the queue.

Data Structures Required

Data Structures Required

```
class Node {
    Node *next;
    int type;
}

// Inherited from base class
class rnode extends Node {
    set<Transaction> tlist;
    type = 0; // read node
}

class wnode extends Node {
    Transaction txn;
    type = 1; // write node
}
```

Data Structures Required

```
    class NodeList {
        Node* head;
        Node* tail;

        // Methods
        addReadNode(Transaction txn);
        addWriteNode(Transaction txn);
        deleteReadNode(Transaction txn);
        deleteWriteNode(Transaction txn);
        getHead();
        getTail();
    }
```

Data Structures Required

```
    class DataItem {
        int value;
        NodeList* nodeList;
        set<int> readList;
        set<int> writeList;
        mutex lock;

        addRead(Transaction t);
        addWrite(Transaction t);
        deleteRead(Transaction t);
        deleteWrite(Transaction t);
    }
```

Data Structures Required

```
class Transaction {
    int tid;
    int threadId;
    status; // init ongoing
}
```

Additionally, we will also declare conditional variables for each thread.

```
atomic<int> conditinals[num_threads];
```

These will later be changed to conditional variables to avoid spinning on atomic variables. These will be accessed by the list data item to notify that the transaction which a thread is executing has become head node and that it can now execute.

Data Structures Required

```
        class WaitsForGraph {
            set<int> vertices; // = transactions
            vector<vector<int>> adjacencyList; // = conflicts

            // Methods
            addOperation(int transId, int ind, int op);
            // transactionId; dataItem; read=0, write=1
            detectCycle();
            garbageCollect();
        }
```

Scheduler Algorithm

Read operations

```
bool read(transaction t, int index, int *localVal) {
    bool permission = WFG.addOperations(t, index, 0); // 0 ->
        read op
    if(permission == 0) {
        t.status = aborted;
        return false;
    }
    shared[index].addRead(t);
    while(conditional[t->threadId] != true) { ; // spinning,
        will be changed }
    localVal = shared[index].value;
    shared[index].deleteRead(t);
    return true;
}
```

```
void addRead(Transaction t) {
    this->lock.lock(); // Lock data item
    this->nodeList->addReadNode(t);
    this->lock.unlock();
}
```

```
void addReadNode(Transaction t) {
    if(this->nodeList->getHead() == nullptr) {
        Node *r = new rnode();
        head = r;
        tail = r;
    }
    if(this->nodeList->getTail().type == 0) {
        tail->tlist.insert(t);
        if(this->nodeList->getTail() == this->nodeList->getHead
            ()) {
            conditional[t->threadId] = true;
        }
    } else {
        Node *r = new rnode();
        r.tlist.insert(t);
        tail->next = r;
        tail = r;
    }
    this.readList.add(t->tid)
}
```

```
void deleteRead(Transaction t) {
    this->lock.lock(); // Lock data item
    this.nodeList.deleteReadNode(t);
    this->lock.unlock();
}
```

```
void deleteReadNode(Transaction t) {
    // If it is here, it means it is head. It will be in the set
         of the tlist of head.
    auto iter = this.head.tlist->find(t);
    this.head.tlist.erase(t);

    if(head.tlist.size() == 0) {
        prev = head;
        head = head->next;
        delete prev;

        // Definitely next should be write node
        conditional[head.transaction->threadId] = true;
    }
}
```

Write Operations

```
    bool write(transaction t, int index, int localVal) {
        bool permission = WFG.addOperations(t, index, 1); // 1
            -> write op
        if (permission == 0) {
            t.status = aborted;
            return false;
        }
        shared[index].addWrite(t);
        while(conditional[t->threadId] != true) { ; }
        shared[index].value = localVal;
        shared[index].deleteWrite(t);
        return true;
    }
```

```
    void addWrite(Transaction t) {
        this->lock.lock(); // Lock data item
        this->nodeList->addWriteNode(t);
        this->lock.unlock();
    }
```

```
void addWriteNode(Transaction t) {
    if(this->nodeList->getHead() == nullptr) {
        Node *w = new wnode();
        head = w;
        tail = w;
        conditional[t->threadId] = true;
    } else {
        Node *w = new wnode();
        w.txn = t;
        tail->next = w;
        tail = w;
    }

    this.writeList.add(t->tid)
}
```

```
void deleteWrite(Transaction t) {
    this->lock.lock(); // Lock data item
    this.nodeList.deleteWriteNode(t);
    this->lock.unlock();
}
```

```
void deleteWriteNode(Transaction t) {
    // If it is here, it means it is head.
    prev = head;
    head = head->next;
    delete prev;

    // Next might be read/write nodes
    // read -> iterate through all threadIds of transactions and
         signal
    // If write node, just signal the transaction of the node
    // conditional[head.transaction->threadId] = true;
}
```

TO-DO

- Consider garbage collection for the wait-for graph to prevent unbounded growth.
- Explore lock-free implementations for data item lists to minimize contention.