Programming Assignment 2: Pseudocode
CS5280

Darpan Gaur
CO21BTECH11004

MVTO-gc

variables

```
class transaction
{
    mutex tLock // lock for transaction
    int id; // transaction id
    // transaction status:  0: active, 1: commit, 2: abort
    int tStatus;
    vector<int> localmem; // local memory
    set<int> read_set;
    set<int> write_set;
}
class item
{
    mutex itemLock // lock for item
    int val; // value of item
    set<int> read_list;
    set<int> W_TS; // write timestamp
    set<pair<int, int>> R_TS; // read timestamp (t_id, item_ts)
}

// scheduler variables
mutex sch_lock; // lock for scheduler
vector<item> items; // vector of items
vector<transaction> tList; // vector of transactions
```

begin_trans

```
begin_trans()
{
    // returns the id for the transaction
    lock(sch_lock);
    int id = idCounter++;
    // create a new transaction
    transaction t= new transaction(id);
    // initialize the variables
    unlock(sch_lock);
    return id;
}
```

read(i, x, l)

```
read(i, x, l)
{
    // i is the transaction id
    // x is the variable to be read
    // store value of x in l
    lock(items[x]->itemLock); // lock the item
    lock(i->tLock); // lock the transaction
    ts = -1;
    for (auto k : items[x]->W_TS) {
        if (k < i->id && k > ts) {
            ts = k;
        }
    }
    if (ts == -1) {
        i->status = 2; // abort the transaction
        unlock(i->tLock);
        unlock(items[x]->itemLock);
        return -1;
    }
    if (i->status == 2) {
        items[x].read_list.erase(i->id);
        unlock(i->tLock);
        unlock(items[x]->itemLock);
        return -1;
    }
    // update the read timestamp
    items[x]->R_TS.insert(make_pair(i->id, ts));
    *l = items[x].val; // read the value
```

```
    i->read_set.insert(x); // update the read set

    unlock(i->tLock);
    unlock(items[x]->itemLock);
    return 0;
}
```

write(i, x, l)

```
write(i, x, l)
{
    // i is the transaction id
    // x is the variable to be written
    // l is the value to be written
    lock(items[x]->itemLock); // lock the item
    lock(i->tLock); // lock the transaction

    for (auto (j, k) : items[x]->R_TS) {
        if (j > i->id && k < i->id) {
            i->status = 2; // abort the transaction
            unlock(i->tLock);
            unlock(items[x]->itemLock);
            return -1;
        }
    }
    // update the write timestamp
    items[x]->W_TS.insert(i->id);
    i->localmem[x] = l; // write the value
    i->write_set.insert(x); // update the write set

    items[x]->read_list.insert(i->id); // update the read list

    unlock(i->tLock);
    unlock(items[x]->itemLock);
    return 0;
}
```

try_commit(i)

```
try_commit(i)
{
```

```
    // i is the transaction id
    lock(i->tLock); // lock the transaction
    if (i->status == 2) {
        for (auto x : i->read_set) {
            lock(items[x]->itemLock); // lock the item
            items[x]->read_list.erase(i->id);
            unlock(items[x]->itemLock); // unlock the item
            return -1;
        }
    }
    for (auto x : i->write_set) {
        lock(items[x]->itemLock); // lock the item
        items[x]->val = i->localmem[x]; // write the value
        unlock(items[x]->itemLock); // unlock the item
        unlock(i->tLock); // unlock the transaction
    }
    i->status = 1; // commit the transaction

    garbage_collect(); // call garbage collect
    unlock(i->tLock); // unlock the transaction
    reutrn 0;
}
```

0.1 garbage_collect

```
garbage_collect()
{
    lock(sch_lock); // lock the scheduler
    // get the min active transaction id
    int min_id = INT_MAX;
    for (auto t : tList) {
        if (t->status == 0 && t->id < min_id) {
            min_id = t->id;
        }
    }
    // get max write timestamp less than min_id
    int max_ts = -1;
    for (auto item : items) {
        for (auto k : item->W_TS) {
            if (k < min_id && k > max_ts) {
                max_ts = k;
            }
        }
```

```
    }
    // delete all W_TS less than max_ts
    for (auto item : items) {
        for (auto k : item->W_TS) {
            if (k < max_ts && tList[k]->status != 0) {
                item->W_TS.erase(k);
            }
        }
    }
    // delete all R_TS less than max_ts
    for (auto item : items) {
        for (auto (j, k) : item->R_TS) {
            if (k < max_ts && tList[j]->status != 0) {
                item->R_TS.erase(make_pair(j, k));
            }
        }
    }
    unlock(sch_lock);
    return 0;
}
```

free_trans(i)

```
free_trans(i) {
    delete localMem
    delete read_set
    delete write_set
    remove i from read_list
}
```