Project5

Concurrency Control

Before doing project 5..

- We allow you to use C++ features, not only C.
- In this project, you need to control multiple threads using mutex or condition variables (pthread API is enough)
  - pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock
  - pthread_cond_wait, pthread_cond_signal

Before doing project 5..

- You need to clarify the terms in database first.
  - Lock: Protects the logical contents in database from other transactions. (what you have to implement in this project!)
  - Latch: Protects the physical data structures from other threads.

    (think about pthread_mutex_t)

Lock Manager

- Your database systems are not yet supporting transaction.
- Implement transaction APIs that can support '**I**solation' and '**C**onsistency' using your own lock manager.
- Your lock manager should provides:
  - Conflict-serializable schedule for transactions
  - Strict-2PL (avoid cascading aborts)
  - Deadlock detection (abort the transaction if detected)
  - Row-level locking with shared(S)/exclusive(X) mode

Project Specification

➢Your library should provide those APIs for wrapping operations in a transaction.

- **int begin_trx();**
  - Allocate transaction structure and initialize it.
  - Return the unique **transaction id** if success, otherwise return 0.
  - Note that transaction id should be unique for each transaction, that is you need to allocate transaction id using mutex or atomic instruction, such as __sync_fetch_and_add().
- **int end_trx(int tid);**
  - Clean up the transaction with given tid (transaction id) and its related information that has been used in your lock manager. (Shrinking phase of strict 2PL)
  - Return the completed transaction id if success, otherwise return 0.

Project Specification

➢Also, your library should provide two APIs, **db_find (for read)** and **db_update (for write)** operations that can be wrapped in a transaction.

- **int db_find(int table_id, int64_t key, char* ret_val, int trx_id)**
  - Read values in the table with matching key for this transaction which has its id **trx_id**.
  - return 0 (SUCCESS): operation is successfully done and the transaction can continue the next operation.
  - return non-zero (FAILED): operation is failed (e.g., deadlock detected) and the transaction should be aborted. Note that all tasks that need to be arranged (e.g., **releasing the locks that are held on this transaction**, **rollback of previous operations**, etc…) should be completed in db_find().

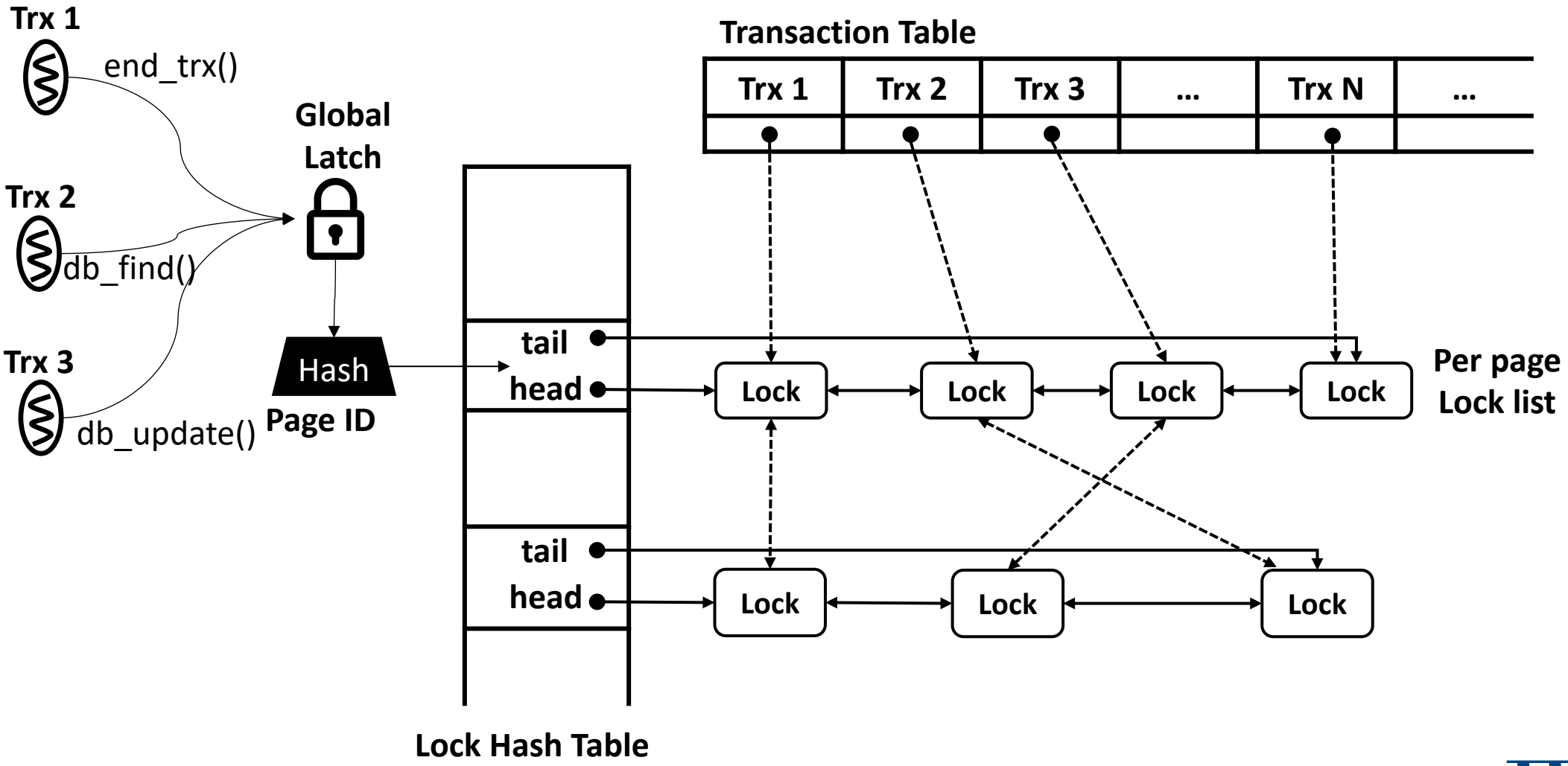
Project Specification

- **int db_update(int table_id, int64_t key, char* values, int trx_id)**
  - Find the matching key and modify the values, where each value (column) never exceeds the existing one.
  - return 0 (SUCCESS): operation is successfully done and the transaction can continue the next operation.
  - return non-zero (FAILED): operation is failed (e.g., deadlock detected) and the transaction should be aborted. Note that all tasks that need to be arranged (e.g., **releasing the locks that are held on this transaction**, **rollback of previous operations**, etc…) should be completed in db_update().

➢ Note that, in this project, you don't have to support db_insert() or db_delete() working in a transaction that may require structure modifications on b+tree.

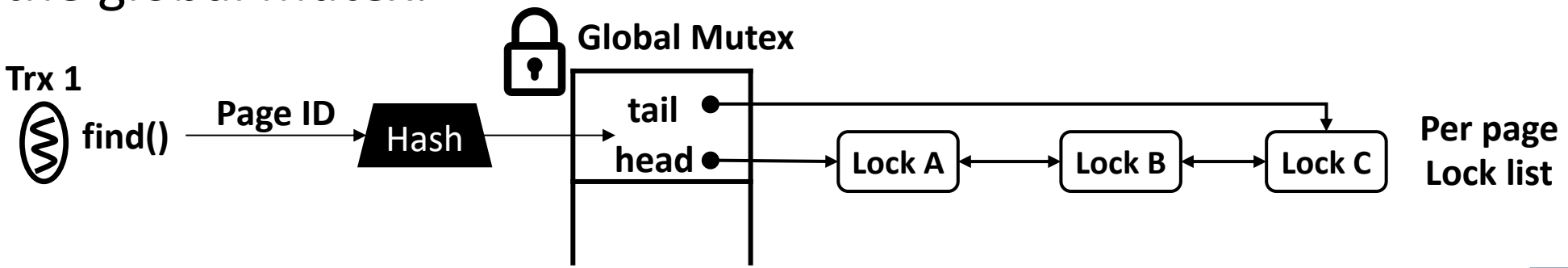
➢ We will first populate the database with db_insert() and then run transactions in our test.

Lock Manager Overview



Lock Manager Implementation

- You need to implement a lock hash table (where each entry can be accessed based on the **page id**)
- Each entry of lock hash table must contain the list of lock objects that are hashed to the bucket.
- Operation which tries to acquire/release a lock needs to acquire a **global lock hash table latch first**. That means any operation which tries to insert or delete lock objects in a lock hash table must be protected by the global mutex.



Lock Manager Implementation

- There are 2 lock modes in each lock object: **{shared(S) | exclusive(X)}** where db_find() requires a lock with S mode while db_update() requires a lock with X mode before performing its operation.
- A lock object should contain detail information enough to figure out if the transaction can proceed or go to sleep or should be aborted. (It's up to your own design!)
- Hint: For detecting deadlock, a lock object can have a backpointer of the transaction who is holding this lock, which also contains the list of locks that this transaction has. (It's up to your design)

Object Structure Example

- Note that this data structure is just for heads-up (not mandatory)

```
struct lock_t {
  int table_id;
  int record_id; // or key
  enum lock_mode mode; // SHARED, EXCLUSIVE
  trx_t* trx; // backpointer to lock holder
  … // up to your design
};
```

```
struct trx_t {
  int trx_id;
  enum trx_state state; // IDLE, RUNNING, WAITING
  list<lock_t*> trx_locks; // list of holding locks
  lock_t* wait_lock; // lock object that trx is waiting
  … // up to your design
};
```

Lock Manager Implementation

- Transaction that follows ***strict-2PL*** can release locks it is holding after all operations are performed. We can do this during end_trx().
- Also, if there is a problem during the operation, transaction should release locks it is holding for **aborting** the transaction. This should be done in db_find() or db_update() operations.

Buffer Management Issues

- Suppose two threads are trying to read or update different records in the same page, where there is no buffer page in memory yet read from disk.
- Your buffer management should also support this concurrent accesses on a buffer page.
- One buffer page (managed by your own buffer control block) should have its **page latch**. Also, your buffer manager needs a **global buffer manager latch**.
- For the concurrent accesses on buffer page, one thread must **(1) acquire a global buffer manager latch** first, read the page from disk if needed, **(2) acquire the page latch** for accessing the buffer page, **(3) release the global buffer manager latch** and operates (read or write) on the buffer page.
- In this manner, we can avoid multiple threads who try to read/write the buffer page at the same time.

Buffer Management Issues

- Acquired page latch is released when the operation on the page (read or update the record in page) is done. That is, only one thread accesses and modifies the buffer page in a quite short time.
- Concurrent access on the same page from multiple threads is protected based on the spin-waiting latches.
- The transaction which is operating on the physical page must have been granted the logical lock object from the lock manager before.
- **Be careful!! Do not let the running transaction sleep(wait) holding with the page latch**, since other transaction who try to access the page will be blocked and can't wake up the slept transaction. (Don't be confused about lock and latch)
- pthread_rwlock_t is for better performance among concurrent read operations.

Milestone & Deadline

- Milestone 1
  - Design your lock manager and express it on Wiki.
  - Implement begin_trx() and end_trx() functions.
  - Deadline : Nov 19 11:59pm
- Milestone 2
  - Implement lock manager and submit a report including your implement, design, issues, etc.
  - Deadline : Dec 5 11:59pm