6.826 Principles of Computer Systems	Fall Semester, 1992

Lecture 13: October 28, 1992

Lecturer: William Weihl *Scribe: Quinton Zondervan*

1 Handouts

Handout 32 (*Concurrent Transactions*).

2 Addendum to last week's lecture

Last week we talked about the sequential transaction crash recovery method, which is the best known method today. Lots of other methods exist, but after 15 years of experience, this one has proven to be the best one so far.

3 Concurrent transactions

Transactions deal with both concurrency and recovery. This concept has been adopted from database systems. Crash recovery is accomplished by discarding any transaction that did not complete before the crash.

In a concurrent system, we could make each external routine atomic, but this has certain disadvantages. For example consider a banking system that looks as follows:

```
Module Bank
  deposit << ... >>
```

In this system:

- We have to know in advance what all the atomic operations should be.
- We cannot extend the set of operations later (transfer, audit, etc.), without changing the synchronization used in the module.

The transaction mechanism allows us to abstract out to an interface and implement the synchronization and atomicity mechanisms. This interface is implemented through 4 commands: **Begin, Do, Commit, Abort**. This makes the applications much simpler.

```
  Applications written in terms of transactions
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  Transactional support (make [Begin ... Commit] look atomic)
```

4 Transactional Support Interface

The system needs to identify each transaction. This is accomplished using tid's. Some systems hide the tid's from the programmer, in other systems and in our specification, tid's are explicit. Begin returns a new tid, and Do, Commit and Abort are supplied with a tid.

In concurrent systems, Do and Commit can both raise the exception *aborted*. This is necessary for example to recover from deadlock (by aborting some transactions), or to implement optimistic synchronization, in which the actions are performed, and if during the Commit the system detects an inconsistency, it aborts the commit.

The module should exhibit the following behavior:

1. Committed transactions should be atomic and persistent.
2. Aborted transactions should appear to have no effect (as if they never happened).

Atomic in this context means:

- **Serializable** A set of transactions is serializable if running them concurrently is the same as if they ran in some arbitrary sequential order.
- **Externally consistent** The apparant serial order of the transactions must be consistent with their real time order. Thus if T1 finishes before T2 starts, then T1 must occur before T2 in the sequence.
- **Recoverability** (or crash atomicity) Either all the effects of a transaction are visible, or none are.
- **Persistence** The effects of committed transactions survive a crash.

Example of Serializable vs. Externally consistent:

```
               T2:write x
T1:Read x          ...
  Write y          ...

T3:Read z
   Write w
                   ...
                 read w
```

These transactions are serializable as T3, T2, T1. However, this is not externally consistent, since T3 must occur after T1. A system that requires both cannot allow this sequence of executions. For example, it could delay T1 until T2 commits, etc. In order to relate the state of the database to the state of the threads accessing it, we need external consistency.

Crash and abort require failure atomicity (recoverability). Only committed transactions can have any effects on the system.

5 Aborted Transactions

What happens to aborted transactions, and why should we care?

If transactions have external side effects (i.e. they modify the real world), we could have a problem! For example, consider an ATM machine. Should it modify the balance before or after giving you the money? If it does it before, and then crashes, you get screwed. If it gives you the money, and crashes before it gets a chance to update your balance, the bank gets screwed. Guess how it's done!

The basic problem is that the system cannot maintain consistency between the real world and the database at all times. Internal inconsistencies could cause strange real world interactions. Even when a transactions has been aborted, we may not be able to undo its real world effects. Therefore, we do have to place constraints on the aborted transactions.

6 Implementation techniques

This section covers the two implementation techniques for the spec described in handout 32.

6.1 Two-phase Locking

This method was first formalized in handout 21.

In 2PL, usually there is a read lock and a write lock, allowing multiple reads but only a single writer. Once a thread has released a lock, it cannot acquire any new ones. This means that there exists some time window during which the thread holds the locks to all the objects it accesses (Figure 1). All operations in the transaction can be pushed into this window without affecting other transactions. Thus the transaction can be serialized as having occured at any point in this window. Since this point is between the `begin` and the `end` of the transaction, it also guarantees external consistency.

Deadlock solution: abort the deadlocked threads.

6.1.1 Cascading Aborts

If T1 is aborted, and T2 read something written by T1, T2 must be aborted as well, and so on. This is known as cascading aborts. To avoid this, T2 is not allowed to read any data written by T1, until T1 commits.

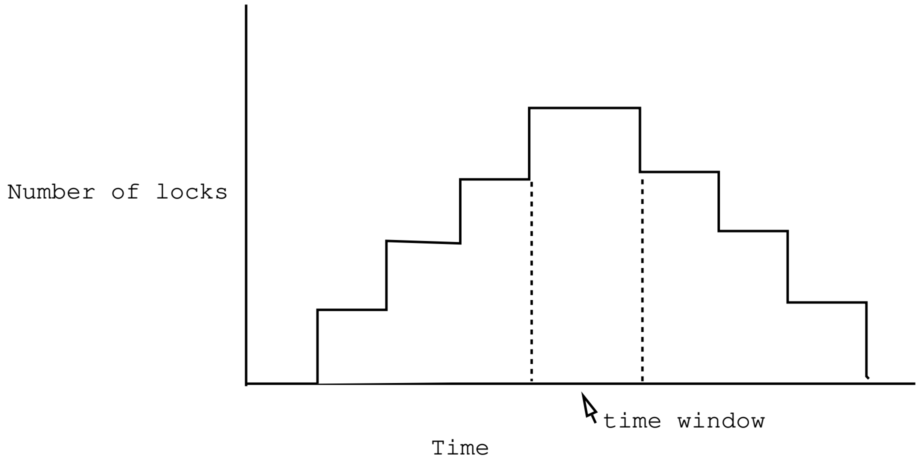

Figure 1: Lock acquisition in 2pl

6.1.2 Strict 2PL

In strict 2PL, the transaction holds all locks until after the commit or abort. This avoids cascading aborts.

6.1.3 Logical locking

Logical locking means associating a lock mode with each action, and requiring two lock modes to conflict whenever their associated actions fail to commute. To make this work, we need to be careful about recovery. In particular, *logical logging* is needed for logical locking. When an action is performed, you must acquire a lock for it as well as for its undo operation, because otherwise, it may not be possible to perform undo later (if blocked by a lock held by another transaction). If you do *value logging* (which means that the undo action just restores the old value), you have to get a write lock for the undo, which eliminates any benefit of logical locking. To get additional concurrency out of logical locking, you need to use undo actions that "invert" the forward actions — e.g., the undo for increment simply does a decrement.

6.2 Multi-version Time Stamps and Other Schemes

Each transaction is given a time stamp when it starts. The system keeps multiple, time stamped versions of each data item. See Handout 32 for more on this subject. There exist hybrid schemes between 2PL and time-stamped methods, but these were not discussed in class. Other schemes include optimistic techniques, which validate consistency at commit time.