Transactions

March 14, 2001

What is a transaction?

- A set of operations that must behave as a single operation
- e.g., { read account balance, add/subtract amount, write new account balance }
- Objectives - Consistency, Fault Tolerance, Performance
- An interface for transactions must implement atomicity

What is atomicity?

- An operation is atomic if you cannot determine that it is composed of a series of steps
  - e.g., Intermediate results not observable
- However, operations that appear atomic at one layer may not appear atomic at others
  - e.g., (define prof (cons 'shai 'simonson)) (set-car! prof 'tom)
    - set-car! is atomic at the Scheme level, but can involve dozens of machine operations
  - If one machine op fails, want (cdr prof) to be either 'shai or 'tom, but not 'thai.

Recoverability

- A sequence of steps such that once initiated, either completes or backs out
  - backs out - the effects of any of the steps that were executed are not observable
  - i.e., “do it once or not at all”
- Recoverable action - Charging an appliance at Wal-mart to your credit card (you can return it)
- Non-recoverable action - Humpty Dumpty - breaking an egg, burning a document, dispensing cash from an ATM

Example: File Logging

- Goal: Make disk writes appear to be atomic with respect to system failures
  - e.g., if the power fails, no partial writes
- Approach: write() returns only after the file system has logged a specific write and written a “done” token
  - The log is a "buffer", on the disk, containing writes that need to be put into their proper place
- Once a write to a file is logged, the file system will not allow access to the file until the logged write is written
- If writing to the log is interrupted, then write() returns an error or exception (if it can)

Transactions - Sample Interface

- Four pieces of a simple interface - Begin, End, Commit, Abort
  - Begin and End denote the set of operations to be implemented as an atomic operation
  - Commit - Completes the transaction - all its effects can now be observed
  - Abort - Backs out - any intermediate effects reversed

ATM - Withdraw(acct_id, amount)

```
begin transaction
acct_token := Open_Account(acct_id, cert, timeout)
if acct_token = 0 {
  abort;
  return could_not_open_account_exception(acct_id)
 }
bal := Read_Balance(acct_token)
set new_bal := bal - amount
if new_bal < 0 then {
   Close_Account (acct_token); abort;
   return negative_bal_exception(acct_id, amount)
   }
Set_Balance (acct_token, new_bal)
Close_Account (acct_token)
commit
end transaction
Dispense_Cash (amount)
```

Bank Account, cont.

- To implement:
  - Need mutual exclusion on an Open_Account(acct_id)
    - No concurrent read or writes
    - Use locks
  - Need ordering
    - e.g., if a Write_Balance is logged by a committed call to Withdraw, it must be observable by a subsequent Read_Balance

Locking

- When a transaction runs, it acquires locks on its resources (e.g., using a mutual exclusion mechanism)
- Two-phase locking: Acquire all locks before releasing any
- Strict two-phase locking:
  - A transaction starts only when all its required locks are acquired
  - Locks are held until transaction aborts
  - If transaction commits, locks are held until any updates are completed

Deadlocks

- P1 holds locks that P2 needs to continue, P2 holds locks that P1 needs to continue
- Deadlock Prevention - can reduce performance significantly
  - e.g., atomic acquisition of locks, ordered locks
- Deadlock Detection - timeouts
  - Careful with Livelock - Deadlocks that keep occurring after identical timeouts

Nested Transactions

- A single transaction can be composed of a set of sub-transactions
  - Why? Better Performance, Nicer Abstraction
- Sub-transactions can provisionally commit
- When the transaction commits, provisionally-committed sub-transactions must commit
- When the transaction aborts, all sub-transactions must abort (even provisionally-committed sub-transactions)
- When a sub-transactions aborts, transaction can either abort or try something different