6.033: Fault Tolerance: Introduction to Transactions
Lecture 15
Katrina LaCurts, lacurts@mit.edu

0. Introduction
   - Main goal: build a reliable system out of unreliable components.
   - Last two days: reliability via replication. GFS gave you replication to tolerate machine failures as long as you had some sweeping simplications.
   - Replication lets us mask failures from users..
   - ..But doesn't solve all of our problems. Can't replicate everything.
   - For failures that replication can't handle: need to reason about them and then deal with them. The reasoning is hard.

1. Atomicity
   - Starting today, we want to achieve "atomicity": Atomic actions happen entirely or not at all. (Sometimes this is called "all-or-nothing atomicity" instead of just atomicity)
   - Why atomicity?
     - Will enable sweeping simplifications for reasoning about fault-tolerance. Either the action happened or it didn't; we don't have to reason about an in-between state.
     - Will be realistic for applications
   - Motivating example:

```
// transfer amount from account_a to account_b
transfer(bank, account_a, account_b, amount):
  bank[account_a] = bank[account_a] - amount
  bank[account_b] = bank[account_b] + amount
```

     Crash between the two lines is bad: deducted amount from a, but didn't add it to b.
   - What should have happened? We exposed internal state; instead, all of this should have happened at once, or not at all -> it should have been atomic.
     - Understanding that this code should be atomic comes from understanding what the application is *doing*. What actions need to be atomic depends on the application.

2. Achieving atomicity
   - Attempt 1:
     - Store a spreadsheet of account balances in a single file.
     - Load the file into memory, make updates, write back to disk when done.
     - (see slides for code)
   - If system crashes in the middle? Okay - on reload, we'll read the file, will look as if transfer never happened.
   - If we crash halfway through writing file? Can't handle that.
   - Golden rule: never modify the only copy.

3. Achieving atomicity with shadow copies
   - New idea: Write to a "shadow copy" of the file first, then rename the file in one step.
     - (see slides for code)
   - If we crash halfway through writing the shadow copy? Still have intact old copy.
   - This makes rename a "commit point"
     - Crash before a commit point => old values
     - Crash after a commit point => new values
     - Commit point itself must be atomic action
   - But then rename itself must be atomic
     - Why didn't we try to make write_accounts an atomic action in the first attempt? Because it would be very hard to do; you'll see after we make rename atomic why this was the right choice

4. Making rename atomic
   - Shadow copies are good. How do we make rename atomic?
   - What must rename() do?
     1. Point "bank_file" at "tmp_bankfile"'s inode
     2. Remove "tmp_bankfile" directory entry
     3. Remove refcount on the original file's inode
   - (see slides for code)
   - Is this atomic?
     - Crash before setting dirent => rename didn't happen
     - Crash after setting dirent => rename happened, but refcounts are wrong
     - Crash while setting dirent => very bad; system is in an inconsistent state.
   - So two problems:
     1. Need to fix refcounts
     2. Need to deal with a crash while setting the dirent

5. Single-sector writes
   - Deal with the second problem by preventing it from happening: setting the dirent involves a single-sector write, which the disk provides as an atomic action.
     - The time spent writing a sector is small (remember: high sequential speed).
     - A small capacitor suffices to power the disk for a few microseconds (this capacitor is there to "finish" the right even on power failure)
     - If the write didn't start, there's no need to complete it; the action will still be atomic.

6. Recovering the disk
   - Other combinations of incrementing and decrementing refcounts won't work (see slides). Mostly because, even if we increment before every applicable action, and decrement after the actions, we could crash between the actions and the increments/decrements.

   - Solution: recover the disk after a crash
     - Clean up refcounts
     - Delete tmp files
     - (see slides for code)
   - What if we crash during recover? Then just run recover again after that crash.

7. Shadow copies: a summary
   - Shadow copies work because they perform updates/changes on a copy and automatically install a new copy using an atomic operation (in this case, single-sector writes)
   - But they don't perform well
     - Hard to generalize to multiple files/directories
     - Require copying the entire file for even small changes
     - Haven't even dealt with concurrency

8. Transactions
   - Speaking of concurrency: we're going to want another thing.
   - Isolation
     - Run A and B concurrently, and it appears as if A ran before B or vice versa.
   - Transactions: powerful abstraction that provides atomicity *and* isolation.
     - We've been trying to figure out how to provide atomicity. Our larger goal is to provide both abstractions, so that we can get transactions to actually work.
   - Examples:

```
T1                      T2
begin                   begin
transfer(A, B, 20)      transfer(B, C, 5)
withdraw(B, 10)         deposit(A, 5)
end                     end
```

     - "begin" and "end" define start and end of transaction
     - These transactions are using higher-level steps than before. T1, e.g., wants the transfer *and* the withdraw to be a single atomic action.

9. Isolation
   - Isn't isolation obvious? Can't we just put locks everywhere?
     - (see slides for code)
   - Not exactly
     - This solution may be correct, but will perform poorly. Very poorly for the scale of systems that we're dealing with now.
       - We don't *really* want only one thread to be in the transfer() code at once
     - Locks inside transfer() won't provide isolation for T1 and T2; those would need higher-level locks.
     - Locks are hard to get right: require global reasoning
   - So we don't know how to provide isolation yet. But we'll get

there eventually. We will *use* locks, just in a much more systematic way that you've seen until now, and in a way that performs well.

10. The Future
    - What we're really trying to do is implement transactions. Right now we have a not-very-good way to provide atomicity, and no way to provide isolation. (unless you count a single, global lock)
    - Next week: We're going to get atomicity and isolation working well on a single machine.
    - Week after: We're going to get transaction-based systems to run across multiple machines.