6.033 Lecture 15 -- Atomicity April 2, 2014

Last week: saw how to build fault tolerant systems out of unreliable parts

Saw how a disk can use checksums to detect a bad block, and how we can use replication of disks to mask bad block errors.

Goal today is to start talking about how we can make a a multi-step action look like a single, ***atomic action***.

Why is this important? Example:
Two bank accounts, balances stored on disk

Transfer routine [show slide: transfer]

```
xfer(A, B, x):
        A = A - x  //writes to disk
                        <--------------- crash!
        B = B + x // writes to disk
```

Lost $x. Note that replication doesn't fix this (unless I can prevent the system from crashing completely.) Replicating everything is impossible:
Have to worry about not just hardware/power failures, but:
- Disk device driver / controller might crash, even with replicated disk
- Python interpreter might crash
- OS kernel might crash

Even if the write to A happens just fine, we've still lost x dollars.

What has happened is we've exposed some internal state of this action that should have been atomic.

What we'd really like is some way to make this action either happen or not happen. E.g., to preserve money.

This is called **"all or nothing atomicity"** -- either it happens or it doesn't.

Related problem: **concurrency**. Suppose we have another function audit that runs concurrently

```
audit(bank):
   sum = 0
   for acct in bank:
      sum = sum + bank[acct]
   return sum
```

[show slide: transfer w/ audit]
These two actions aren't isolated -- they saw each other's intermediate state.

Audit should only have seen $200

Isolation goal: Serial equivalence —> outcome of concurrent actions is equivalent to some serial execution of those actions.

E.g., A & B == A then B or B then A.

Note that we have already seen a way to do this with locks, but that requires complex manual reasoning. For example, suppose a user issues a series of transfer that should be done atomically:

```
transfer(A,B)
transfer(C,D)
transfer(D,A)
```

can't just push locking logic into transfer -- procedure that does transfer has to do locking to provide isolation.

Goal is to develop an abstraction that provides both all-or-nothing atomicity and serial equivalence for concurrent actions.

[slide: two goals]

[slide: transactions]

This abstraction is called "atomic actions" or "**transactions**"

Works as follows:

```
T1                  T2
begin               begin
transfer(A,B,20)    transfer(B,C,5)
debit(B,10)         deposit(A,5)
...                 ....
end                 end
aka "commit"
```

Properties:
- User doesn't have to put explicit locking anywhere.
- All or nothing
- Serial equivalence
- No need to pre-declare the operations -- things become visible when "end" is issued.

Extremely powerful abstraction. Tricky to implement, but makes programmers life easy.

Today -- start talking about implementing all-or-nothing property without isolation (which we will address in a couple of lectures.)

Suppose we've got a file of account balances (e.g., an excel spreadsheet) stored in a single file

Let's just focus on implementing transfer atomically:

Strawman approach — load into memory, make updates, write back when done

```
xfer(bankfile, a, b, amt):
        bank = read_accounts(bankfile)
        bank[a] = bank[a] – amt
        bank[b] = bank[b] + amt
        write_accounts(bankfile)
```

demo:
python transfer.py

<u>Q: what is the problem with this?</u>

might get have way through writing file and crash, then file is in some intermediate state

Golden rule of atomicity: never modify the only copy if you want multiple distinct operations to appear atomic

version 2 ("shadow copy")

```
xfer(bankfile, a, b, amt):
        bank = read_accounts(bankfile)
        bank[a] = bank[a] – amt
        bank[b] = bank[b] + amt
        write_accounts("#bankfile")
        rename("#bankfile", bankfile)
```

demo:
python transfer.py rename

Fixes problem, as long as rename is atomic

E.g., after crash, either have old version or new version, but not neither version, or something else.

can we implement atomic rename using unix ops link and unlink:

```
        unlink(file)
                                <--- crash!  (file disappears)
        link(fnew, file)
        unlink(fnew)
```

So what do we need to do?
[show slide: file system data structures]

```
 First try at rename(fnew,file):
   file's dirent gets fnews inode #
                                        <— crash here?  (refcount too small!)
   decref(file's original inode)
   remove fnew's dirent
```

What happens if we crash after modifying file's dirent?
Potentially problematic: two names point fnew's inode, but refcount is 1. Problem is that we don't know which reference is the correct one.

What if we increment refcount before, and decrement it afterwards?

```
   rename(fnew,file):
     newino = lookup(fnew)
     oldino = lookup(file)

     incref(newino)
                               <— crash!  newino's refcount is 2, should be 1
    file's dirent  gets newino  <— commit point
                               <—— crash!  no file points to oldino, its refcount = 1
     decref(oldino)
     remove new's dirent
                               <— crash!  refcount newino is 2, should be 1
     decref(newino)
```

We never have too few refcounts, but still might have too many.

We say the **commit point** is modifying file's dirent — after this op completes, the rename has happened (modulo fixing up refcounts)

[Show slides — rename in action]

What if we crash during the commit point writing to the dirent?

assume a single disk write is all-or-nothing -- disks mostly implement this (e.g., using a capacitor that "finishes" a write)

If we crashed before commit: extra refcount on inode 2.
If we crashed after commit: extra refcount on inode 1.

In both cases: directory entry for "newfile" can be removed.
(before ==> “nothing”, after ==> “all”)

```
"recover":  (restart processing to ensure all or nothingness)
        scan fs, fix refcounts
        if exists(fnew)
                remove(fnew) // file is either old or new
```

why did shadow copy work?

performed updates / changes on a copy

atomically installed new copy using a single all-or-nothing operation (sector write)

this is the key ==> need to “install” new data with a single write, creating a well-defined commit point

general approach for atomic operations:

- write two copies of the data
- switch to new copy (atomically)
- remove old copy

some way towards our vision of atomicity, but it doesn’t solve all of our problems:

1) have to copy the whole file every time, even if we only modified a few records

2) only works for one file -- what if changes span multiple files?
   (e.g., both customer and accounts table)

3) one operation at a time ==> no concurrency

Going to address these limitations next time
Next time we will develop a general atomicity preserving scheme that works for multiple objects.

Doesn't require us to copy all blocks of an object -- only those we modify.

Idea is to write a log of each change we make, reflecting the state of the object that was changed before and after the change. Can then use this log to go from any intermediate state to the "before" state or the "after" state.

E.g.,: Database A, Database B; A10 v0 B7 v0
Before:

    A10 v0 B7 v0

After:

    A10 v1 B7 v1
    Log:
        W(A10 v0,A10 v1)
        W(B7 v0, B7 v1)

(Log records contain both old and new copies)
Suppose crash, come back, and find disk has A10 v1, B7 v0
Can use the log to go back to both being version 0 or both being version 1