6.033 Lecture 16 -- Atomicity April 6, 2009

Sam Madden

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

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

Notice that 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 two people run:

```
A = 10;  B  = 20
U1                      U2
----                    ---
transfer(A,B,5)         transfer(A, B, 5)
```

suppose A = A-X is actually

```
        R0 = read(A)
        R0 = R0 - X
```

write(A, R0)

Outcomes:

A = 0, B = 30 (correct)

U1	U2
RA	
	RA
	WA
WA	

A = 5 (not correct!)

These two actions weren't isolated -- they saw each other's intermediate state.

Should have only allowed A = 0, B = 30.

Isolation goal: 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.

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.

Most common system that uses these techniques is a database. Rather than representing data as abstract files, it represents data as tables with records.

For example:

Accounts:

Customers:

...

Can write transactions that read and modify a mix of multiple tables, e.g.

- complex transfer ops between accts or
- delete a customer and all of his accounts.

...

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 database balances (e.g., an excel spreadsheet) stored in a single file

Imagine that the file is loaded into memory, and that the "save" operation works as follows:

```
save(file):
        for each record r in file:
                if r is changed
                        write b to file
```

Q: what is the problem with this?

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

version 2 ("shadow copy")

```
save(file):
        copy(file,fnew)
        for each record r in fnew:
                if r is dirty, write r o fnew
        move(fnew, file) //rename  --- commit point!
```

requires an atomic rename operation

after crash, either have old version or new version

atomic rename:

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

actual:

```
        link(fnew, file)
                                <-- crash!  (two copies of file!)
        unlink(fnew)
```

link command is the "commit point" -- after it happens, atomic action is guaranteed to finish in it's entirety (the "all" part of "all or nothing")

File system state on disk

directory block:

name	inode#
file	1
fnew	2

inode 1:
blocks: A, B, C
refcount: 1

inode 2:
blocks D, E, F
refcount: 1

link:

<---- crash! //both file and fnew exist, but different
(CP) modify directory block //both fnew and file point to new file
<---- crash! //refcount wrong
update refcount

unlink:

<---- crash! //both fnew and file exist, refcount wrong
modify directory block
<---- crash! // only file exists, refcount wrong
update refcount

assume a single disk write is atomic -- disks mostly implement this (e.g., using a capacitor that "finishes" a write)

will see the details of how files systems can ensure this property without this assumption in Thursday's recitation (or see the notes -- section 9.2 discusses); basic idea is to write two copies of the block, return the one that is valid using checksums

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

Q: when is the commit point? just after the modify directory block in link

some way towards our vision of atomicity; problems:

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

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

  - could arrange all the databases you want to edit in a directory, atomically rename new directory

  - requires fixes directory structure beforehand

- unclear how concurrency works. can achieve isolation if we only allow one user to access at a time, but that's very restrictive.

(we would rather allow different uses to concurrently access different parts of a file)

Going to address these limitations next time, but before we do some key features of this approach:

- we write two copies of the data
- switch to new copy
- remove old copy

All of our atomicity schemes are going to work like this --
Golden rule of recoverability:

NEVER OVERWRITE ONLY COPY OF DATA.

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