6.033 Spring 2016
Lecture #15

• When replication fails us
• Atomicity via shadow copies
• Transactions
• Isolation
**high-level goal:** build reliable systems from unreliable components

this is difficult because reasoning about failures is difficult. we need some abstractions that will let us simplify.
atomicity

an action is atomic if it happens completely or not at all. if we can guarantee atomicity, it will be much easier to reason about failures

we’ll have to think about the consequences of the action happening or not happening, but not about the action partially happening
transfer (bank, account_a, account_b, amount):
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount

problem: account_a lost amount dollars, but account_b didn’t gain amount dollars
transfer (bank, account_a, account_b, amount):
   bank[account_a] = bank[account_a] - amount
   bank[account_b] = bank[account_b] + amount

solution: make this action atomic. ensure that we complete both steps or neither step.
quest for atomicity: attempt 1

```
transfer (bank_file, account_a, account_b, amount):
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts(bank_file) ← crash!
```

**problem:** a crash during `write_accounts` leaves `bank_file` in an intermediate state
quest for atomicity: attempt 2
(shadow copies)

transfer (bank_file, account_a, account_b, amount):
  bank = read_accounts(bank_file)
  bank[account_a] = bank[account_a] - amount
  bank[account_b] = bank[account_b] + amount
  write_accounts(“tmp_file”)
  rename(“tmp_file”, bank_file) ← crash!

problem: rename must itself be atomic
           (so that we can only fail before or after it, not during)
directory data blocks
   filename “bank_file” -> inode 1
   filename “tmp_file” -> inode 2

inode 1: // old data
   data blocks: [..]
   refcount: 1

inode 2: // new data
   data blocks: [..]
   refcount: 1

need to:
1. point “bank_file”’s dirent at inode 2
2. delete “tmp_file”’s dirent
3. remove refcount on inode 1
directory data blocks
filename “bank_file” -> inode 2
filename “tmp_file” -> inode 2

inode 1: // old data
  data blocks: [..]
  refcount: 1

inode 2: // new data
  data blocks: [..]
  refcount: 1

rename(tmp_file, orig_file):
  tmp_inode = lookup(tmp_file) // = 2
  orig_inode = lookup(orig_file) // = 1

orig_file dirent = tmp_inode
directory data blocks
filename “bank_file” -> inode 2

inode 1: // old data
  data blocks: [..]
  refcount: 1

inode 2: // new data
  data blocks: [..]
  refcount: 1

rename(tmp_file, orig_file):
  tmp_inode = lookup(tmp_file)  // = 2
  orig_inode = lookup(orig_file)  // = 1

orig_file dirent = tmp_inode
remove tmp_file dirent
directory data blocks

filename “bank_file” -> inode 2

inode 1: // old data
  data blocks: [..]
  refcount: 0

inode 2: // new data
  data blocks: [..]
  refcount: 1

rename(tmp_file, orig_file):
  tmp_inode = lookup(tmp_file)  // = 2
  orig_inode = lookup(orig_file)  // = 1

  orig_file dirent = tmp_inode
  remove tmp_file dirent
  decref(orig_inode)
directory data blocks
filename “bank_file” -> inode 1
filename “tmp_file” -> inode 2

inode 1: // old data
  data blocks: [..]
  refcount: 1

inode 2: // new data
  data blocks: [..]
  refcount: 1

rename(tmp_file, orig_file):
  tmp_inode = lookup(tmp_file)
  orig_inode = lookup(orig_file)

  orig_file dirent = tmp_inode
  remove tmp_file dirent
  decref(orig_inode)

← crash! ⚡
rename didn’t happen
directory data blocks

filename “bank_file” -> inode 2
filename “tmp_file” -> inode 2

inode 1: // old data
  data blocks: [..]
  refcount: 1

inode 2: // new data
  data blocks: [..]
  refcount: 1

rename(tmp_file, orig_file):
  tmp_inode = lookup(tmp_file)
  orig_inode = lookup(orig_file)

orig_file dirent = tmp_inode
remove tmp_file dirent
decref(orig_inode)

← crash! ⚫
rename happened,
but refcounts are wrong
directory data blocks
filename “bank_file” -> inode ?
filename “tmp_file” -> inode 2

inode 1: // old data
    data blocks: [..]
    refcount: 1

inode 2: // new data
    data blocks: [..]
    refcount: 1

rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)
    orig_inode = lookup(orig_file)

    orig_file dirent = tmp_inode ← crash!💥
    remove tmp_file dirent
decref(orig_inode)

crash during this line seems bad..
but won’t happen; single-sector writes
are themselves atomic
we’re trying to make a sequence of actions atomic using shadow copies: write to a temporary file, and then rename it to the original.

rename itself must be atomic, and we’ve almost got that working — thanks in part to atomic single-sector writes — but our refcounts aren’t quite correct.
directory data blocks
filename “bank_file” -> inode 2
filename “tmp_file” -> inode 2

inode 1: // old data
  data blocks: [..]
  refcount: 1

inode 2: // new data
  data blocks: [..]
  refcount: 1

rename(tmp_file, orig_file):
  tmp_inode = lookup(tmp_file)
  orig_inode = lookup(orig_file)

  orig_file dirent = tmp_inode
  remove tmp_file dirent
  decref(orig_inode)

  crash! 🌧
rename happened, but refcounts are wrong
directory data blocks
    filename “bank_file” -> inode 1
    filename “tmp_file” -> inode 1

inode 1: // old data
    data blocks: [..]
    refcount: 1

inode 2: // new data
    data blocks: [..]
    refcount: 1

rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)
    orig_inode = lookup(orig_file)
    incref(tmp_inode)
    orig_file dirent = tmp_inode
    decref(orig_inode)
    remove tmp_file dirent
    decref(tmp_inode)

this approach still won’t work.
refcounts will never be too small, but could still be too large after a crash.
solution: **recover** from failure.
clean things up.

```python
recover(disk):
    for inode in disk.inodes:
        inode.refcount = find_all_refs(disk.root_dir, inode)
        if exists("tmp_file"):
            unlink("tmp_file")
```
atomicity: solved?

not quite. shadow copies perform poorly even for a single user and a single file. and we haven’t even talked about concurrency…

isolation
(second abstraction)

if we guarantee isolation, then two actions A1 and A2 will appear to have run **serially** even if they were executed concurrently (i.e., A1 before A2, or vice versa)
transactions: provide atomicity and isolation

T1
begin
  transfer(A, B, 20)
  withdraw(B, 10)
end

T2
begin
  transfer(B, C, 5)
  deposit(A, 5)
end

atomicity: T1 and T2 will each appear to have run to completion, or not at all

isolation: when run concurrently, it will appear as if T1 and T2 were run sequentially (serially)
atomicity and isolation — and thus, transactions — make it easier to reason about failures (and concurrency)

right now you have a sense of how we might make transactions atomic (via shadow copies), provided there was no concurrency and we didn’t care about performance
transfer (bank_file, account_a, account_b, amount):
    acquire(lock)
    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    write_accounts("tmp_file")
    rename("tmp_file", bank_file)
    release(lock)

couldn’t we just put locks around everything?
   (isn’t that what locks are for?)
transfer (bank_file, account_a, account_b, amount):
  acquire(lock)
  bank = read_accounts(bank_file)
  bank[account_a] = bank[account_a] - amount
  bank[account_b] = bank[account_b] + amount
  write_accounts("tmp_file")
  rename("tmp_file", bank_file)
  release(lock)

this particular strategy will perform poorly
  (would force a single transfer at a time)

also, locks sometimes require global reasoning,
  which is messy
  (eventually, we’ll incorporate locks, but in a systematic way)
goal: to implement transactions, which provide atomicity and isolation, while not hindering performance

atomicity → shadow copies. work, but perform poorly and don’t allow for concurrency

isolation → ?

eventually, we also want transaction-based systems to be distributed: to run across multiple machines
• **Transactions** provide **atomicity** and **isolation**, both of which make it easier for us to reason about failures because we don’t have to deal with intermediate states.

• **Shadow copies** are one way to achieve atomicity. The work, but perform poorly: require copying an entire file even for small changes, and don’t allow for concurrency.