Lecture #15: Atomicity, Isolation, Transactions
introducing abstractions to make fault-tolerance achievable
our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high.

RAID allows us to recover from single disk failures on one machine.

The high-level process of dealing with failures is to identify the faults, detect/contain the faults, and handle the faults. in lecture, we will build a **set of abstractions** to make that process more manageable.
atomicity

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

\begin{verbatim}
transfer (bank, account_a, account_b, amount):
  bank[account_a] = bank[account_a] - amount ← crash!
  bank[account_b] = bank[account_b] + amount
\end{verbatim}

\textbf{problem:} \texttt{account_a} lost \texttt{amount} dollars, but \texttt{account_b} didn't gain \texttt{amount} dollars
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

solution: make this action atomic. ensure that the system completes both steps or neither step.

current quest: update the bank transfer code to ensure that this action is atomic
transfer (bank_file, account_a, account_b, amount):

    bank = read_accounts(bank_file)
    bank[account_a] = bank[account_a] - amount ← crash! *
    bank[account_b] = bank[account_b] + amount
    write_accounts(bank_file)

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.

Current quest: update the bank transfer code to ensure that this action is atomic.

Idea: write to a file so that a crash in between lines 2 and 3 has no effect.
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.

Current quest: Update the bank transfer code to ensure that this action is atomic.

Idea: Write to a file so that a crash in between lines 2 and 3 has no effect.

**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.
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) ← crash!
rename(tmp_file, bank_file)

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.

current quest: update the bank transfer code to ensure that this action is atomic

idea: write to a temporary file so that a crash in between lines 2 and 3 has no effect, and neither does a crash during a write
atomicty

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

```python
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:** a crash during `rename()` potentially leaves `bank_file` in an intermediate state

**idea:** write to a temporary file so that a crash in between lines 2 and 3 has no effect, and neither does a crash during a write

current quest: update the bank transfer code to ensure that this action is atomic
atomicty

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

**solution:** make `rename()` atomic

**making rename()** atomic more feasible than making `write_accounts()` atomic; we’ll see why as we go along

**current quest:** update the bank transfer code to ensure that this action is atomic

**idea:** write to a temporary file so that a crash in between lines 2 and 3 has no effect, and neither does a crash during a write
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)  // = 2
    orig_inode = lookup(orig_file) // = 1

    // point orig_file’s dirent at inode 2
    // delete tmp_file’s dirent
    // remove refcount on inode 1

we’re in an interlude, working on making rename atomic. this is
the bank transfer code, which we’ll eventually return to

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)

current quest: ensure that rename is atomic, so that our approach to the
bank transfer code works

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
relevant data structures

<table>
<thead>
<tr>
<th>directory entries</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>filename “bank_file”</td>
<td>-&gt; inode 2</td>
</tr>
</tbody>
</table>

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
delete tmp_file dirent
decref(ori_inode)

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’re in an interlude, working on making rename atomic. this is the bank transfer code, which we’ll eventually return to

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)

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works
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)

  crash! ✗

  (here, or anywhere above this)

  it's as if rename didn't happen

directory entries
  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

relevant data structures

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

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)

we're in an interlude, working on making rename atomic. this is the bank transfer code, which we'll eventually return to

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works
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  (here, or anywhere after this)
    decref(orig_inode)

    rename happened, but refcounts might be wrong

we’re in an interlude, working on making rename atomic. this is
the bank transfer code, which we’ll eventually return to

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)

current quest: ensure that rename is atomic, so that our approach to the
bank transfer code works

relevant data structures

<table>
<thead>
<tr>
<th>directory entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename “bank_file” -&gt; inode 2</td>
</tr>
<tr>
<td>filename “tmp_file” -&gt; inode 2</td>
</tr>
</tbody>
</table>

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

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
**directory entries**

- filename “bank_file” -> inode 1
- filename “tmp_file” -> inode 2

<table>
<thead>
<tr>
<th>inode</th>
<th>data blocks</th>
<th>refcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[..]</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>[..]</td>
<td>1</td>
</tr>
</tbody>
</table>

**rename**(*tmp_file*, *orig_file*):

1. `tmp_inode = lookup(tmp_file)`  // = 2
2. `orig_inode = lookup(orig_file)`  // = 1

- `orig_file` dirent = `tmp_inode` ← crash! ✎
- remove `tmp_file` dirent
- `decref(orig_inode)`

*crash during* this line seems bad...
but is okay because single-sector writes are themselves atomic

**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're in an interlude, working on making `rename` atomic. This is the bank transfer code, which we'll eventually return to.

```python
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)
```

Current quest: ensure that `rename` is atomic, so that our approach to the bank transfer code works.
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  (here, or anywhere after this)
decref(orig_inode)      rename happened,  
but refcounts might be wrong

current quest: ensure that rename is atomic, so that our approach to the 
bank transfer code works

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're in an interlude, working on making rename atomic. this is 
the bank transfer code, which we'll eventually return to

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)
rename(tmp_file, orig_file):
    tmp_inode = lookup(tmp_file)  // = 2
    orig_inode = lookup(orig_file)  // = 1
    incref(tmp_inode)
    orig_file dirent = tmp_inode
    decref(orig_inode)
    remove tmp_file dirent
    decref(tmp_inode)

    problem: this is a mess, and is still incorrect

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works
**solution**: recover from failure
(clean things up)

recover(disk):
  for inode in disk.inodes:
    inode.refcount = find_all_refs(disk.root_dir, inode)
  if exists(tmp_file):
    unlink(tmp_file)

having a recovery process means that we don’t have to worry about getting everything completely correct before the failure happens; we have a chance to clean things up afterwards

**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’re in an interlude, working on making rename atomic. this is the bank transfer code, which we’ll eventually return to

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)

current quest: ensure that rename is atomic, so that our approach to the bank transfer code works
isolation refers to how and when the effects of one action (A1) are visible to another (A2). In lecture, we will aim to get a high level of isolation, where A1 and A2 appear to have executed serially, even if they are actually executed in parallel.

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.

isolation deals with concurrency, and we’ve seen that. Couldn’t we just put locks around everything? (Isn’t that what locks are for?)

```python
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)
```
isolation refers to how and when the effects of one action (A1) are visible to another (A2). In lecture, we will aim to get a high level of isolation, where A1 and A2 appear to have executed serially, even if they are actually executed in parallel.

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.

transer (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)

isolation deals with concurrency, and we’ve seen that. Couldn’t we just put locks around everything? (Isn’t that what locks are for?)

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

Locks sometimes require global reasoning, which is messy eventually, we’ll incorporate locks, but in a systematic way
atomicity and isolation — and thus, transactions — make it easier to reason about failures (and concurrency)

**transactions:** provide atomicity and isolation

**Transaction 1**
begin
  transfer(A, B, 20)
  withdraw(B, 10)
end

**Transaction 2**
begin
  transfer(B, C, 5)
  deposit(A, 5)
end

**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

**isolation**

isolation refers to how and when the effects of one action (A1) are visible to another (A2). *in lecture*, we will aim to get a high level of isolation, where *A1 and A2 appear to have executed serially*, even if they are actually executed in parallel.
our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high.

The high-level process of dealing with failures is to identify the faults, detect/contain the faults, and handle the faults. in lecture, we will build a **set of abstractions** to make that process more manageable.

RAID allows us to recover from single disk failures on one machine.
our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high.

**transactions** — which provide **atomicity** and **isolation** — make it easier for us to reason about failures.

our job in lecture is to understand how a system *implements* these two abstractions.

how do our systems guarantee atomicity? how do they guarantee isolation?

**atomicity**: we have this working for one user and one file via *shadow copies*, but they perform poorly.

**isolation**: we don’t really have this yet (coarse-grained locks perform poorly; fine-grained locks are difficult to reason about).
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.