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

```python
transfer (bank, account_a, account_b, amount):
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
```
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

```python
def transfer(bank, account_a, account_b, amount):
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    crash!
```
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.

**Problem:** account\_a lost amount dollars, but account\_b didn’t gain amount dollars.

```python
transfer (bank, account\_a, account\_b, amount):
    bank[account\_a] = bank[account\_a] - amount ← crash!
    bank[account\_b] = bank[account\_b] + amount
```
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.
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

**transfer** (bank, account_a, account_b, amount):

\[ \text{bank}[\text{account}_a] = \text{bank}[\text{account}_a] - \text{amount} \quad \text{← crash!} \]

\[ \text{bank}[\text{account}_b] = \text{bank}[\text{account}_b] + \text{amount} \]

**solution:** make this action atomic. ensure that the system completes both steps or neither step.
**transfer** *(bank_file, account_a, account_b, amount)*:

```Python
bank = read_accounts(bank_file)
bank[account_a] = bank[account_a] - amount
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.
transfers (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)

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)

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! ✗

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

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(tmp_file)
    rename(tmp_file, bank_file)
```
```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) ← 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.
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)

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.
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! ❌

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
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

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

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.
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):**

```python
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! ❌
```

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

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
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
**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
def 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.
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.
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.
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

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

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

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

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

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

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

relevant data structures

<table>
<thead>
<tr>
<th>directory entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename “bank_file” -&gt; inode 1</td>
</tr>
<tr>
<td>filename “tmp_file” -&gt; inode 2</td>
</tr>
</tbody>
</table>

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

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

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
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>

<table>
<thead>
<tr>
<th>INODE 1: // old data</th>
<th>INODE 2: // new data</th>
</tr>
</thead>
<tbody>
<tr>
<td>data blocks: [..]</td>
<td>data blocks: [..]</td>
</tr>
<tr>
<td>refcount: 1</td>
<td>refcount: 1</td>
</tr>
</tbody>
</table>

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’s dirent
  // remove refcount on inode 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.

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
// remove refcount on inode 1

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

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.

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.

---

**relevant data structures**

Directory entries

<table>
<thead>
<tr>
<th>Filename</th>
<th>Inode</th>
</tr>
</thead>
<tbody>
<tr>
<td>bank_file</td>
<td>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
dirent
remove tmp_file dirent
decref(orig_inode)
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)

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)

it's as if rename didn't happen

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)

relevant data structures

<table>
<thead>
<tr>
<th>directory entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename “bank_file” -&gt; inode 1</td>
</tr>
<tr>
<td>filename “tmp_file” -&gt; inode 2</td>
</tr>
</tbody>
</table>

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

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

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: 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
crash! ✱
remove tmp_file dirent
(decode, or anywhere after this)
decref(orig_inode)

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

relevant data structures

directory entries
  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

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

    orig_file dirent = tmp_inode
crash! ❗️
    remove tmp_file dirent
    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

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

orig_file dirent = tmp_inode ← crash! ✗
remove tmp_file dirent
de_unref(orig_inode)

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>
</tr>
</thead>
<tbody>
<tr>
<td>filename &quot;bank_file&quot; -&gt; inode ?</td>
</tr>
<tr>
<td>filename &quot;tmp_file&quot; -&gt; inode 2</td>
</tr>
</tbody>
</table>

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 ← crash! ❗️
remove tmp_file dirent
decref(orig_inode)

   crash during this line seems bad..

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
relevant data structures

<table>
<thead>
<tr>
<th>filename</th>
<th>inode</th>
</tr>
</thead>
<tbody>
<tr>
<td>bank_file</td>
<td>?</td>
</tr>
<tr>
<td>tmp_file</td>
<td>2</td>
</tr>
</tbody>
</table>

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 ← crash! ⚠
remove tmp_file dirent
decref(orig_inode)

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

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)

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.
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

    crash! ☹

relevant data structures

directory entries
    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

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
   incref(*tmp_inode*)
   *orig_file* dirent = *tmp_inode*
   decref(*orig_inode*)
   remove *tmp_file* dirent
   decref(*tmp_inode*)

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*)

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>

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

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

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
    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

directory entries
    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

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)
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**

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)
```
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

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)
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)

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
**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.

```python
def 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.

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

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)

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

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.

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)

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.
atomicity and isolation — and thus, transactions — make it easier to reason about failures (and concurrency)

**transactions:** provide atomicity and isolation

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>transfer(A, B, 20)</td>
<td>transfer(B, C, 5)</td>
</tr>
<tr>
<td>withdraw(B, 10)</td>
<td>deposit(A, 5)</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

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.

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.
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
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?
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.
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.

we haven't covered how one would use shadow copies in general (e.g., outside of the world of banking). and we won’t; next time, we’ll work on a scheme that is superior in every way