6.1800 Spring 2024

Lecture #16: Atomicity, Isolation, Transactions

introducing abstractions to make fault-tolerance achievable

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

you have an exam tomorrow

there are a lot of things you can use to study, all on the website

- lecture outlines, slides
- recitation notes
- practice exams

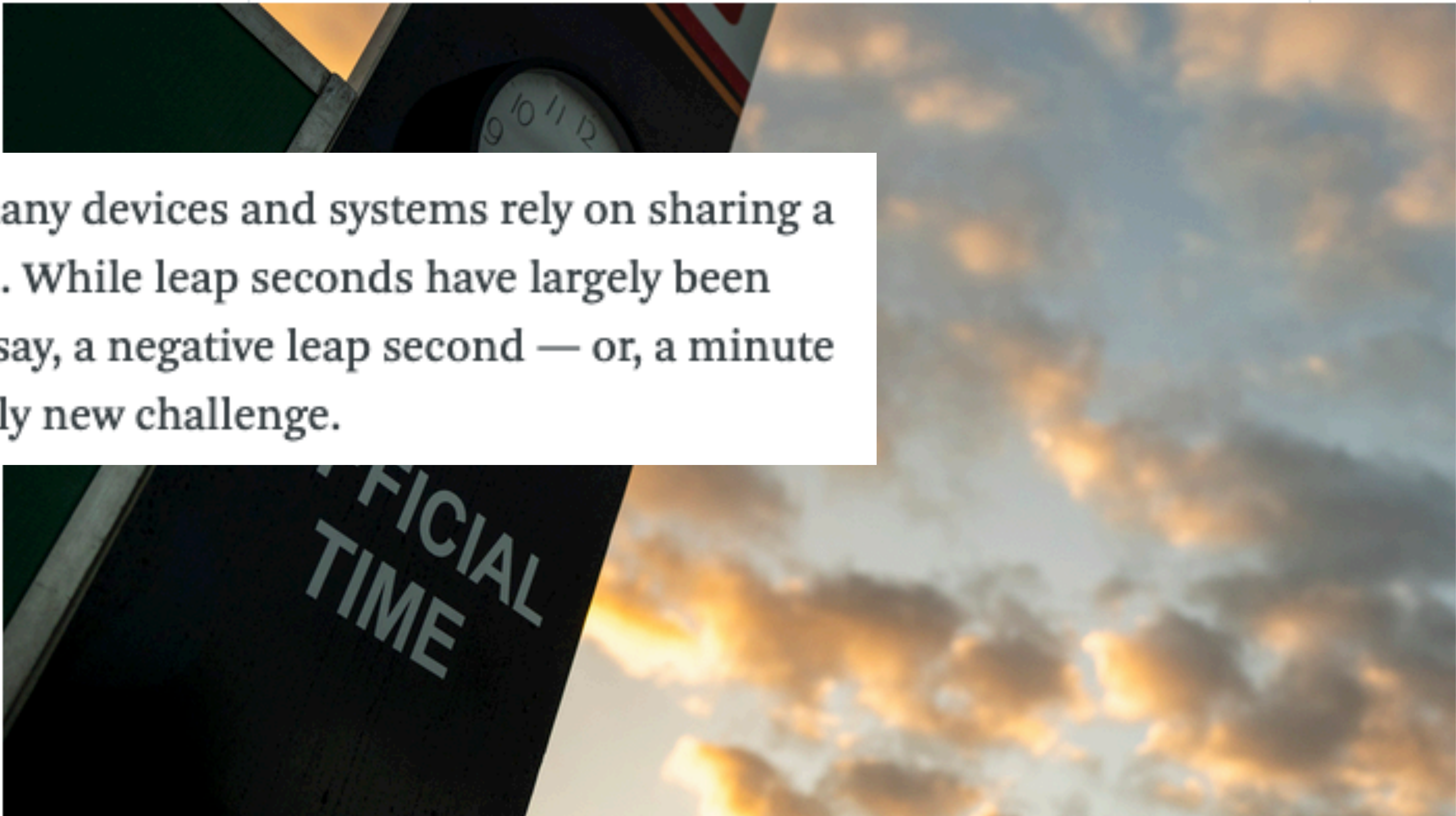
the exam is open book but not open Internet. you will turn your network devices off during the exam. download everything you might need ahead of time.

you all can do well on this exam, get some sleep tonight

6.1800 in the news

Negative leap second: Climate change delays unusual step for time standard

March 30, 2024 | By Bill Chappell



In our technologically interconnected era, many devices and systems rely on sharing a certain awareness of precisely what time it is. While leap seconds have largely been absorbed into current mechanisms, experts say, a negative leap second — or, a minute with only 59 seconds — could pose an entirely new challenge.

https://www.wbur.org/npr/1241674216/climate-change-time-negative-leap-second

6.1800 in the news

They were created as a way to reconcile deviations between traditional astronomical time and the newer international reference based on atomic clocks, known as Coordinated Universal Time or UTC. It's a process that for years has been complicated by variations in the Earth's rotation.

"By the 1960s, Earth was and had been decelerating, and so rotating more slowly than in the nineteenth century, which defined the atomic second," Agnew writes.

The first leap second was added in 1972. In the first decades, it became nearly an annual process. In the past 23 years, scientists have added only four leap seconds, according to Agnew.

"Since 1972, irregularities in Earth's movement have called for 27 leap seconds to be added — at irregular intervals and with a maximum of only 6 months' notice each time," said Patrizia Tavella, director of the Time Department at the International Bureau of Weights and Measures in France, in a discussion published in *Nature* along with Agnew's research.

The current problem, Agnew says, is that Earth's rotation now seems to be gradually getting faster than the established time standard can account for.

https://www.wbur.org/npr/1241674216/climate-change-time-negative-leap-second

6.1800 in the news

And here's where things get a bit weird. Human-induced climate change actually acts to slow down the planet's rotation, Agnew says, because when ice melts at the poles, the planet gets a bit more oblong — wider at the equators — and less spherical. That means Earth spins a little slower, like when an ice skater holds their arms out, rather than pulling them in.

The net result for timekeepers, the new research says, is that climate change seems to have delayed the potential need for a negative leap second, at least for a bit.

https://www.wbur.org/npr/1241674216/climate-change-time-negative-leap-second

6.1800 in the news

what assumptions are built into our systems? what happens when those assumptions are wrong?

Leap seconds have had critics for a long minute, in part because of the havoc they can wreak on things like online reservation and retail systems. A couple years ago, engineers at Meta railed against it, stating, "Introducing new leap seconds is a risky practice that does more harm than good" and should be replaced.

Worth noting: "In 2012, a leap second caused a major Facebook outage, as Facebook's Linux servers became overloaded trying to work out why they had been transported one second into the past," the Data Center Dynamics website noted.

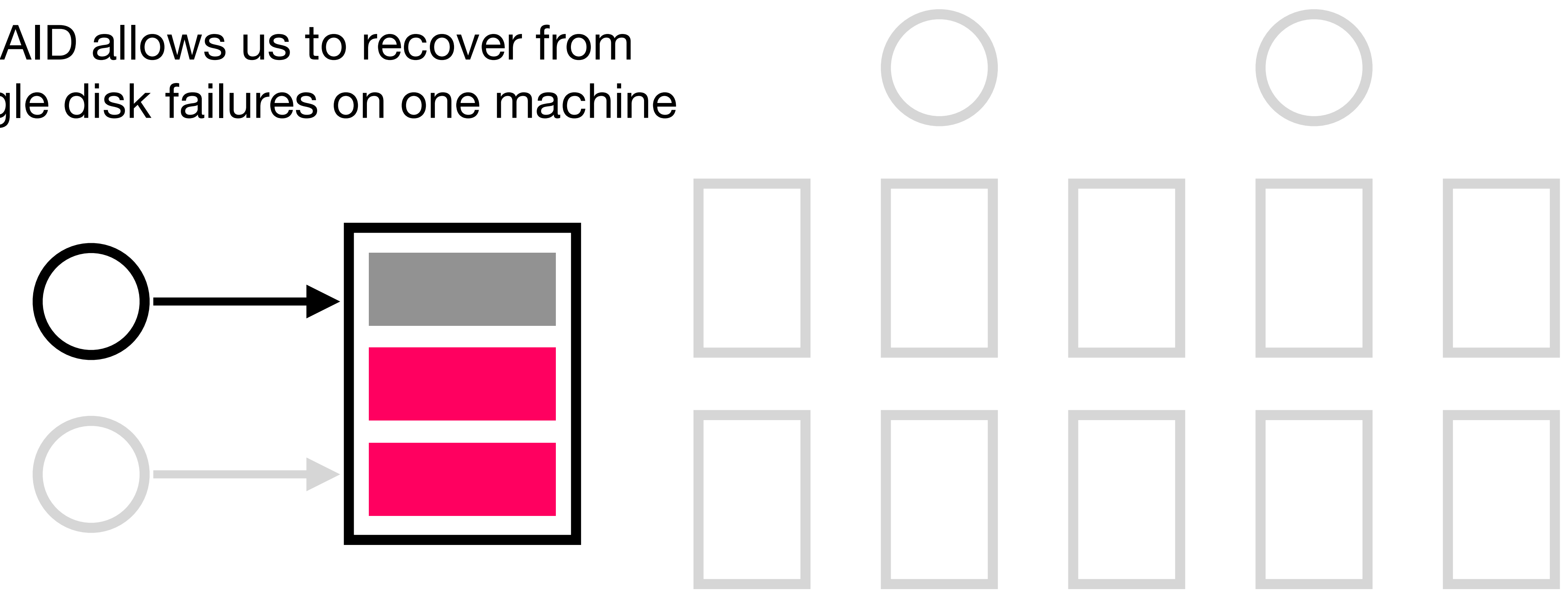
"Many systems now have software that can accept an additional second, but few if any allow for removing a second," Agnew said, "so that a negative leap second is expected to create many difficulties."

Other solutions could present themselves. In 2022, the General Conference on Weights and Measures decided to eliminate the leap second by 2035. The organization could decide to eliminate the potential of a negative leap second sooner than that deadline.

Agnew suggests the groups that determine UTC should adopt a new rule: "never allow" a negative leap second.

https://www.wbur.org/npr/1241674216/climate-change-time-negative-leap-second

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

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

← **crash!** 💥

problem: `account_a` lost `amount` dollars, but `account_b` didn't gain `amount` dollars

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, account_a, account_b, amount):
    bank[account_a] = bank[account_a] - amount    ← crash! 💥
    bank[account_b] = bank[account_b] + amount
```

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

`current quest: update the bank transfer code to make this action 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

```
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   ← crash! 💥
    write_accounts(bank_file)
```

if the system crashes here, upon recovery, it will appear as if the transfer didn't happen at all

because we didn't make any updates to `bank_file`

current quest: update the bank transfer code to make this action *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(bank_file)  ← crash! 💥
```

problem: a crash during `write_accounts()` leaves `bank_file` in an intermediate state

current quest: update the bank transfer code to make this action *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)
```

if the system crashes here, upon recovery, it will appear as if the transfer didn't happen at all

because we didn't make any updates to `bank_file`. we don't read from `tmp_file`, so it's okay if it was left in an intermediate state

current quest: update the bank transfer code to make this action *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

```
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

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 make this action *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! 💥
```

solution: make `rename()` atomic

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

current quest: update the bank transfer code to make this action *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

relevant data structures

```
directory entries
    filename “bank_file” -> inode 1
    filename “tmp_file” -> inode 2

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

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

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)
```

relevant data structures

```
directory entries
   filename "bank_file" -> inode 2

inode 1: // old data             inode 2: // new data
    data blocks: [..]                data blocks: [..]
    refcount: 0                      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)
```

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)
```

relevant data structures

```
directory entries
    filename "bank_file" -> inode 1
    filename "tmp_file" -> inode 2

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

crash! 💥 (here, or anywhere above this)

it's as if rename didn't happen

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)
```

relevant data structures

```
directory entries
    filename "bank_file" -> inode 2
    filename "tmp_file" -> inode 2

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

crash! 💥

(here, or anywhere after this)

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)
```

relevant data structures

```
directory entries
    filename “bank_file” -> inode ?
    filename “tmp_file” -> inode 2

inode 1: // old data            inode 2: // new data
    data blocks: [..]               data blocks: [..]
    refcount: 1                     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..
but is okay because single-sector writes
are themselves atomic**

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)
```

relevant data structures

```
directory entries
    filename “bank_file” -> inode 2
    filename “tmp_file” -> inode 2

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

crash! 💥 (here, or anywhere after this)

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)
```

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)
```

renaming the file — specifically modifying `bank_file`'s directory entry — is the commit point. if the system crashes before the commit point, it's as if the operation didn't happen; if it crashes after the commit point, the operation must complete. the commit point itself must also be 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

```
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*?

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

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.

transactions provide atomicity and isolation

```
Transaction 1          Transaction 2
  begin                  begin
  transfer(A, B, 20)     transfer(B, C, 5)
  withdraw(B, 10)        deposit(A, 5)
  end                    end
```

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

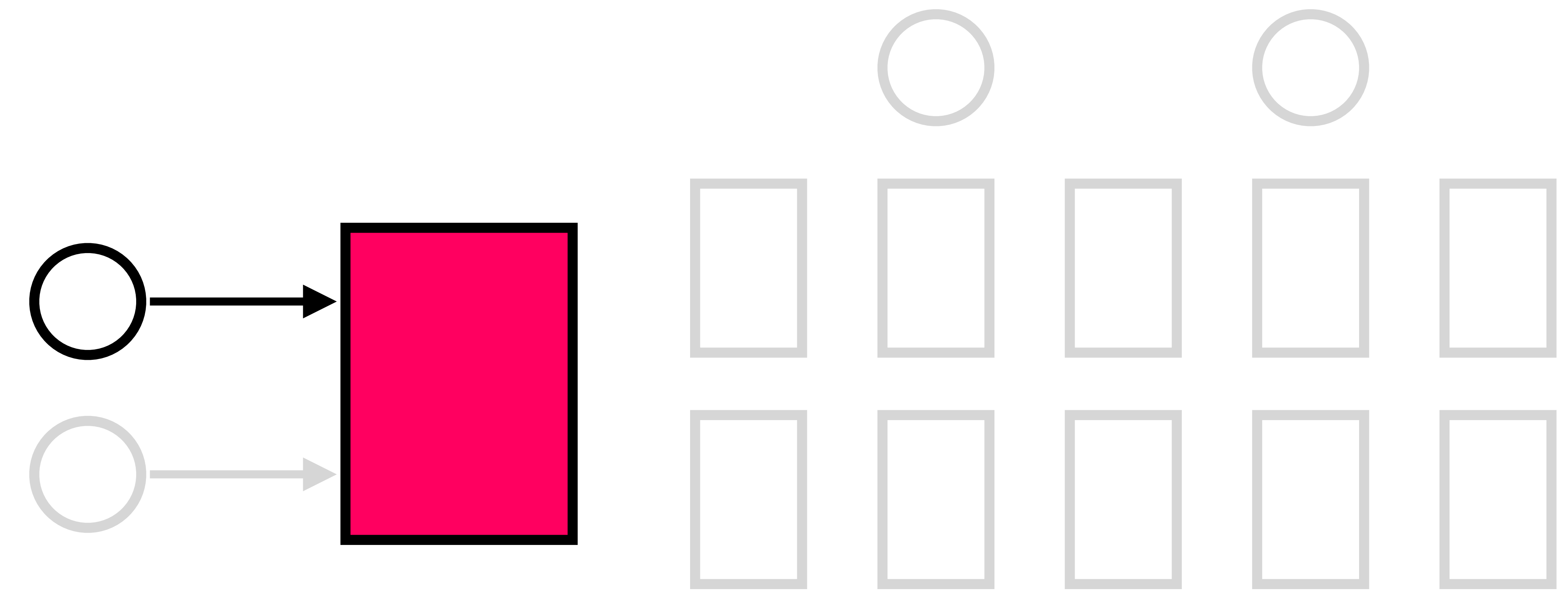
atomicity

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

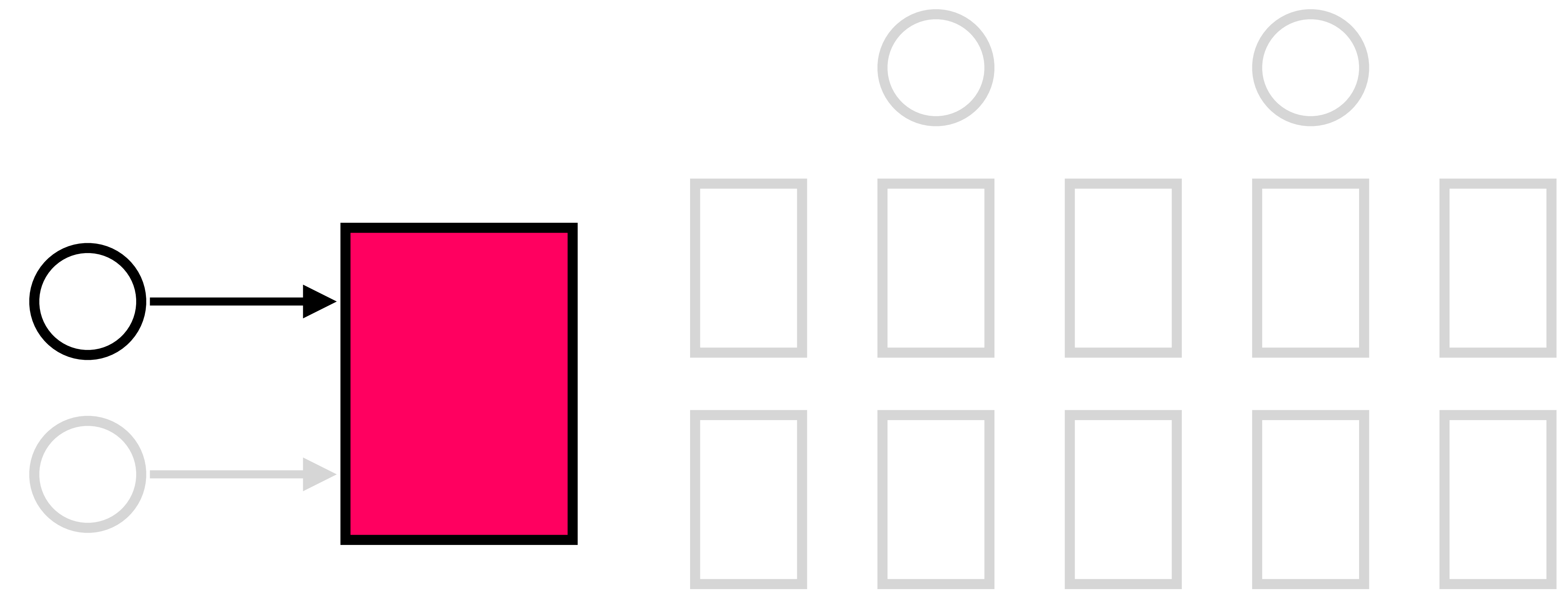
isolation refers to how and when the effects of one action (A1) are visible to another (A2). *in lecture*, we will aim to get a high level of isolation, where **A1 and A2 appear to have executed serially**, even if they are actually executed in parallel.

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

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* transactions.
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.

our main goal for the next few lectures is to *implement* transactions. how do we get the underlying system to provide atomicity and isolation so that this abstraction can exist?

shadow copies are one way to achieve atomicity. they work in certain cases, but perform poorly: — requiring us to copy 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