Our current high-level goal is to build reliable systems from
unreliable components. We've agreed that in order to reason about
failures, we need to provide some abstractions that make things
easier.

The main abstraction we're going for is a transaction. Transactions
give us two things:

1. Atomicity: A transaction happens entirely, or not at all.
2. Isolation: Two transactions, when run in parallel, will appear to
   have been run in sequence (spoiler: we won't really run them in
   sequence; that would perform terribly).

Why are those things good?
- Atomicity means we don't have to worry about our system being in
  some intermediate state. In any failure, the action happened
  or it didn't; it didn't happen halfway.
- Isolation means we can reason about concurrency. This is useful
  even if things *aren't* failing.

So we're trying to implement transactions. Where are we?

- Atomicity: shadow copies. The golden rule of atomicity motivated
  these: never modify the only copy.

  Shadow copies work fine for a single user and a single file, but
  don't perform well (copying the entire file even for small changes).

- Isolation: currently unsolved, unless you consider using one global
  lock for everything (terrible performance).

Today we're going to keep focusing on atomicity, and in particular
providing atomicity while not hindering performance.

Let's start where we left off with shadow copies. Shadow copies *are*
actually used in some applications. They're simple, and an effective
design choice in some cases (some UNIX applications, like text
editors, use them in part because of atomic rename).
But let's look at one more thing we could do with shadow copies first; it'll motivate subsequent techniques.

Aborts + Commits

Abort in case of error

```python
def transfer(bankfile, account_a, account_b, amount):
    bank = read_accounts(bankfile)
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    if bank[account_a] < 0:
        print "Not enough funds"
    else:
        write_accounts("tmp_bankfile")
        rename(tmp_bankfile, bankfile)
```

Here, if account_a ends up with fewer than 0 dollars, the transfer() doesn't happen; it "aborts".

Being able to do this is nice; we can sort of "roll back" to our previous state if something goes on. This notion appears in transactions too, along with one other thing: commits.

```python
def transfer(bankfile, account_a, account_b, amount):
    begin
    bank[account_a] = bank[account_a] - amount
    bank[account_b] = bank[account_b] + amount
    if bank[account_a] < 0:
        abort
    else:
        commit
```

Logging

Motivating Example

But let's move on to improving performance. Specifically, we don't want to write an entire file every time we make one small change. Since we're not concerned with isolation right now, we are just considering one transaction running at a time. Thus, for this lecture, our primary concerns is that if the system crashes mid-transaction, we don't want the effects to be visible after the
Consider our bank account example again.
- Two accounts: A and B.

These are the persistent values that we need to preserve atomicity for. May be helpful to think of them as a database of values, arranged in a table, e.g.:

<table>
<thead>
<tr>
<th>Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

This tabular representation is how most database work.

And then consider the following transactions. We want to run these as all-or-nothing (atomic) actions (and again, for this lecture, we're running them one after the other, not concurrently):

```
begin      // T1
A = 100
B = 50
commit     // At commit: A=100; B=50

begin      // T2
A = A - 20
B = B + 20
commit     // At commit: A=80; B=70

begin      // T3
A = A + 30
--CRASH--
```

Basic idea: Logging

Our basic idea will be to keep a log of all changes, and whether each change commits or aborts. Spoiler: initially, this won't perform very well. We'll fix that by the end of the class.

So what will go into the log?
1. Assign every all-or-nothing action a unique transaction ID
   - Why? Need to distinguish multiple actions in progress at the same time.
2. Two kinds of records:
   (i) UPDATE records: include both the new and old value of some variable.
- We'll see in a bit why we need the old values

(ii) COMMIT/ABORT records: specify whether that action committed or aborted.

- So if you want to be pendantic, three types of records

Here's the log for our previous example:

```
+-------+------+--------+-------+------+--------+-------+
TID    |  T1  |  T1    | T1    | T2    | T2    | T2    | T3    |
OLD    | A=0  | B=0    |       | A=100 | B=50  |       | A=80  |
NEW    | A=100| B=50   | COMMIT| A=80  | B=70  | COMMIT| A=110 |
+-------+------+--------+-------+------+--------+-------+
```

Initially, what seems good about this? For every transaction, we're only writing the values that changed. We don't have to copy over every single piece of data in a file (including values that didn't change).

Using the log to implement atomic persistent storage

So what happens when the program runs? How does this give us atomicity?

1. On begin: Allocate a new transaction ID
2. On write: Append an entry to the log
3. On read: Scan the log looking for the last committed value
   - What if you want to see your own commits? Read the last uncommitted value from your own TID. This is part of the reason why we append an entry for every write instead of adding the new values along with COMMIT.
     In fact, since we're not dealing with concurrent reads right now, we could *almost* just say "read the last value", but that idea will be incorrect on a crash.
4. On commit: Write a commit record
   - Expectedly, writing a commit record is the "commit point" for an action, because of the way read works (look for a commit record). However, writing log records must itself be an atomic action. One approach, from last time, is to make each record fit within one sector.

Another approach would be to put some kind of checksum on each record. Since we're only worried about half-written records when there's a crash, partially-written records are okay to skip over.
because their effects won't be included in any committed transaction anyway.

5. On abort: We could write an abort record. But for the scheme described so far, it's actually okay to do nothing.

6. On recover from crash: Do nothing

Performance

So how well does this perform? We care about three things: write performance, read performance, and recovery performance (how long it takes to recover after a crash):

1. Write performance: should be good. Writes are sequential, instead of random. (Sequential writes are one huge bonus of logging.)

2. Read performance: abysmal. Scan the log for every read! If the value wasn't updated recently, you're going to be scanning for a very long time.

3. Crash recovery: Instantaneous; there's nothing to do.

Improving Read Performance with Cell Storage

So how can we optimize read performance? We'll keep both a log and "cell storage".

The log is just as before. It's authoritative, provides all-or-nothing atomicity. Cell storage provides fast reads, but cannot provide atomicity. It's important to know, though, that are cell storage is *on disk*. This is not an in-memory cache.

Now, updates (writes) will go to the log and to cell storage, and we will read from cell storage.

Initially, we made a big deal about reading the "last committed" value. Which we still want. But remember, we're not dealing with concurrency yet. So if we just read from cell storage, we either have the last committed value, or our current value.

We'll say we "log" an update when it's written to the log, and "install" an update when it's written to cell storage.

So, questions:
1. How do we recover cell storage from an authoritative log after a crash? In particular, you should be concerned with the fact that what was one a single atomic write is now two writes (one to the log and one to cell storage); what if we crash in between?

2. Why did we write the way we did? I.e., why did we write to the log before the cell? Does it matter?

Recovery

We'll focus on recovery first. Let's look at the previous example in our situation. The log still contains the same things, and as we're running we maintain cell storage for A and B.

<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
<th>T2</th>
<th>T2</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</td>
<td>COMMIT</td>
<td>A=80</td>
<td>B=70</td>
<td>COMMIT</td>
<td>A=110</td>
</tr>
</tbody>
</table>

CELL STORAGE
A = 110
B = 70

Except one problem: after crash, A's value in cell storage is wrong. That last action (T3) was aborted, but the changes to A are visible. So we need to repair this in our recovery function.

Remember: since the cell storage is on disk, the value for A did not get wiped by the crash the way it would have if cell storage was in memory.

Good thing we have the log to provide authoritative information!

If we log an update, install it, and then abort or crash, we need to undo that installed update. We'll scan the log, determine what actions aborted ("losers"), and undo them. (Non-losers are "winners".)

This is why our log records contain old and new values.

Why scan backwards? We need to undo newest to oldest. We also need to find the outcome of the action before we decide whether to undo.

What if we crash during recovery? It's fine. recovery() is idempotent: can keep recovering over and over again. If we crash during the undo phase, restarting is okay; will perform the same undo. If we crash while logging aborts, restarting is okay; will duplicate...
Why does the ordering of logging and installing matter? I.e., why did we log before installing?

Because the two together don't have all-or-nothing atomicity. If we crash in between, just one of the two might have taken place.

If we install first, and then log, we have no idea what happened to cell storage or how to fix it. The log can't help us because the log hasn't been written yet. This violates the golden rule of atomicity: never modify the only copy.

Note that logging appends, while installing *overwrites*.

The golden rule of logging, then, is to log an update before installing it. If we crash, the log is authoritative and intact, and can repair cell storage. This protocol is known as "Write-ahead logging".

Performance

Okay, so performance-wise: how are we doing?

1. Writes: probably still okay, although we do write twice (log, then install)
3. Recovery: Not great. Requires scanning the entire log. Scanning the log will take longer and longer as the log grows.

Further Performance Improvements

We can deal with both of those performance issues (writes being okay but not great, recovery being terrible).

Optimization 1: Caching

The first optimization is a cache: we'll defer installing updates to
cell storage by storing them in a cache.

Writes can now be fast: just one instead of two. The hope is that the variable is modified several times in the cache before it's flushed.

Reads will still be fast, because they can go through the cache (in fact, reads might get even faster than they were before).

But, of course, there's an atomicity problem: cell storage (on disk) may be out-of-date.

- Could we have changes that should be in cell storage, but aren't? Yes, if we haven't flushed the latest commits (in fact, we'd expect for this to happen frequently; after all, that's the whole point of the cache).

- Could we have changes that shouldn't be in cell storage, but are? Yes, if we flushed some changes and then aborted.

During recovery, we need to go through and re-apply changes to cell storage.

- UNDO every ABORT (even if it had an explicit record). We don't treat actions with an abort record as "done", because there might be leftover changes from them in cell storage.

- REDO every COMMIT

Optimization 2: Truncation

So we've made writes faster. What about recovery? Our current design requires the log to grow without bound, which is not practical. So we're going to truncate the log.

But is this even possible? Are there parts of the log that can be discarded? Remember, it's supposed to be an authoritative source for us; seems bad to throw part(s) of it away.

So what parts, if any, can be discarded? We must know the outcome of every action that is part of the log, and cell storage must reflect all of those log records (commits, aborts).

Assuming no pending actions, here's how we'll truncate:
- Flush all cached updates to cell storage
- Write a CHECKPOINT record to save our place in the log
- Truncate the log prior to the CHECKPOINT record.

(Often the log is implemented as a series of files, so doing this
amounts to deleting old log files.)

With pending actions, we could delete before the checkpoint and earliest undecided record.

Remember: Once the cache is flushed, cell storage has everything. We assume data there is uncorrupted because we know how to deal with disk failures. The log's purpose is for crash recovery, not for data storage (even though you can reconstruct updates from it).

What happened to ABORT records?

This whole time we've never used an ABORT record. That is in part because we've only been discussing crashes; if you crash mid-transaction, there's no time to write an ABORT record anyway.

But what if not? What if the programmer decides to abort explicitly, without crashing?

We'd need to use the log to find all update records that were part of this action, reset cell storage to old values from those records, and make sure writes during abort are WAL (would be in trouble if we crashed mid-abort).

So could we just skip writing an ABORT record, and pretend we crashed? In terms of correctness, yes: because we undo all actions that weren't completed and redo those that were completed, it's okay to skip.

However, writing ABORT records allows us to skip UNDO'ing aborted transactions. So you can think of ABORT records as an optimization for recovery. They're not strictly needed, but they can be helpful in the recovery process (tell you some things that you *don't* have to deal with).

What about un-UNDO-able actions?

These systems are real, and some things you can't undo: dispensing cash, shipping a package, firing a missile, etc. You can't UNDO these. They also aren't idempotent, so we certainly don't want to REDO them.

But this is a real concern in real systems; it requires special care. One approach is to wait for software that controls an action to COMMIT, and then take the action afterwards, but have some special way to detect if an action already happened (e.g., sensor of amount of cash, or whether missile bay is empty/full).
Parts of the log

Let's recap. Why did we need all those parts and actions?
- ID: Might need to distinguish between multiple actions at the same time (going to be more important when we consider concurrency)
- Undo: roll back losers, in case we wrote to cell storage before the abort/crash
- Redo: apply commits, in case we didn't write to cell storage before commit.

Remember: The log is there to help you reconstruct the data after a crash. It's NOT to meant to be the only authoritative source of where the data is.

Logging in general

Logging is a general technique for achieving atomicity. It's widely used in databases, file systems. You can achieve pretty reasonable performance with logging:
- Writes are always fast (sequential)
- Reads can be fast with cell storage
- Key Idea 1: WAL, makes it safe to update cell storage
- Key Idea 2: Recovery protocol. Undo losers, Redo winners.

Tomorrow in recitation you'll see logging used in a file system.

Where we're headed

So now we've done a pretty good job achieving atomicity with a single user; logging performs much better than shadow copies. And remember, we want atomicity because it makes it easier for us to reason about failures (the action happened or it didn't, but it's not in some intermediate state).

The other thing that makes it easier to reason about failure is isolation, and we're going to deal with that on Wednesday. We want multiple transactions to run at the same time, but appear as if they were run sequentially. "Appear" means, at a very high-level, that the outcome corresponds to what would've happened if they were run
sequentially. We, of course, will run multiple transactions in parallel. Without that, performance would be abysmal.

After Wednesday, we'll spread our transaction-based systems out across multiple machines.