Lecture #17: Atomicity via Logging
superior to shadow copies in almost every way
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)
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**: logging, which is going to provide us with much better performance at the cost of some added complexity.

**isolation**: we don’t really have this yet (coarse-grained locks perform poorly; fine-grained locks are difficult to reason about).
begin  // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin  // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit  // A=80; B=70

begin  // T3
write(A, read(A)+30)
crash!💥

**problem:** after crash, A=110, but T3 never committed

we need a way to revert to A's previous committed value
```
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
```

---

<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></td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>COMMIT</td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>COMMIT</td>
<td>UPDATE</td>
</tr>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</td>
<td>A=100</td>
<td>B=50</td>
<td>A=80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</td>
<td>A=80</td>
<td>B=70</td>
<td>A=80</td>
<td>A=110</td>
<td></td>
</tr>
</tbody>
</table>
let's try to read the value of $A$ from this log

read(log, var):
    commits = []
    // scan backwards
    for record r in log[len(log) - 1] .. log[0]:
        // keep track of commits
        if r.type == COMMIT:
            commits.add(r.tid)
        // find var’s last committed value
        elif r.type == UPDATE and
            r.tid in commits and r.var == var:
            return r.new_value

begin  // T1
write(A, 100)
write(B, 50)
commit  // A=100; B=50

begin  // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit  // A=80; B=70

begin  // T3
write(A, read(A)+30)

commits = [T1]
<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
</tr>
</tbody>
</table>

**let's try to read the value of A from this log**

```python
read(log, var):
    commits = []
    // scan backwards
    for record r in log[len(log) - 1] .. log[0]:
        // keep track of commits
        if r.type == COMMIT:
            commits.add(r.tid)
        // find var's last committed value
        elif r.type == UPDATE and
            (r.tid in commits or r.tid == current_tid) and r.var == var:
            return r.new_value
```

let's try to read the value of A from this log

brief interlude: we're going to change this example slightly, to illustrate one additional point
let's try to read the value of A from this log

```
read(log, var):
    commits = []
    // scan backwards
    for record r in log[len(log) - 1] .. log[0]:
        // keep track of commits
        if r.type == COMMIT:
            commits.add(r.tid)
        // find var’s last committed value
        elif r.type == UPDATE and
            (r.tid in commits or r.tid == current_tid)
            and r.var == var:
            return r.new_value
```

now back to our original example

```
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

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

after a crash, the log is still correct; uncommitted updates will not be read
Katrina LaCurts  lacurts@mit.edu  6.033 2022

begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

<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></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
<td>A=110</td>
</tr>
</tbody>
</table>

read(log, var):
commits = []
// scan backwards
for record r in log[len(log) - 1] .. log[0]:
    // keep track of commits
    if r.type == COMMIT:
        commits.add(r.tid)
    // find var’s last committed value
    elif r.type == UPDATE and
        (r.tid in commits or r.tid == current_tid) and r.var == var:
        return r.new_value

problem: reads can be very slow
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

**problem:** the value of A in cell storage never committed (and so shouldn't be read); we need to repair cell storage

### read(var):
return cell_read(var)

### write(var, value):
log.append(current_tid, “UPDATE”, var, read(var), value)
cell_write(var, value)
<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td></td>
<td>A=100</td>
<td>B=50</td>
<td></td>
</tr>
<tr>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A=100</td>
<td>B=50</td>
<td>A=80</td>
<td>B=70</td>
<td>A=110</td>
</tr>
</tbody>
</table>

**Cell Storage (on disk):**

```
A 80  B 70
```

begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

**Recover (log):**

```
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo
```

```
commits = [T2, T1]
```
<table>
<thead>
<tr>
<th>TID</th>
<th>UPDATE</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</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></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
<td>A=110</td>
</tr>
</tbody>
</table>

**begin** // T1
write(A, 100)
write(B, 50)
**commit** // A=100; B=50

**begin** // T2
write(A, read(A)-20)
write(B, read(B)+20)
**commit** // A=80; B=70

**begin** // T3
write(A, read(A)+30)
crash!

**read(var):**
return cell_read(var)

**write(var, value):**
log.append(current_tid, “UPDATE”, var, read(var), value)
cell_write(var, value)

**recover(log):**
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo

**problem:** read performance is now great, but writes got (a little bit) slower and recovery got (a lot) slower
**read(var):**

```python
if var in cache:
    return cache[var]
else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]
```

**write(var, value):**

```python
log.append(current_tid, update, var, read(var), value)

cache[var] = value
```

**flush():** // called “occasionally”

```python
for var in cache:
    cell_write(var, cache[var])
```

**recover(log):**

```python
commits = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo
```

Suppose we flushed the cache after T1 committed, but have not flushed it since then.
### Cell Storage (on Disk)

<table>
<thead>
<tr>
<th>TID</th>
<th>UPDATE</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</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>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</td>
<td></td>
<td>A=80</td>
<td>B=70</td>
<td></td>
</tr>
</tbody>
</table>

### Cache (Memory)

- A: 100
- B: 50

### Cell Storage

- A: 110
- B: 70

### Code Snippets

**begin**

```
// T1
write(A, 100)
write(B, 50)
commit // A=100; B=50
```

**begin**

```
// T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70
```

**begin**

```
// T3
write(A, read(A)+30)
```

**crash!**

**read(var):**

```python
if var in cache:
    return cache[var]
else:
    # may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]
```

**write(var, value):**

```python
log.append(current_tid, update, var, read(var), value)
cache[var] = value
```

**flush():** // called “occasionally”

```python
cell_write(var, cache[var])
```

**recover(log):**

```python
commit = []
for record r in log[len(log)-1] .. log[0]:
    if r.type == COMMIT:
        commits.add(r.tid)
    if r.type == UPDATE and r.tid not in commits:
        cell_write(r.var, r.old_val) // undo
```
Katrina LaCurts | lacurts@mit.edu | 6.033 2022

<table>
<thead>
<tr>
<th>TID</th>
<th>UPDATE</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</th>
<th>COMMIT</th>
<th>UPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[\text{cell storage (on disk)}\]

<table>
<thead>
<tr>
<th>A 80</th>
<th>B 50</th>
</tr>
</thead>
</table>

\[\text{cache (memory)}\]

```
begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!
```

all other updates were committed; B's value won't ever be changed

\[\text{cell storage (on disk)}\]

\[\text{begin} // T1\]
\begin{itemize}
\item write(A, 100)
\item write(B, 50)
\end{itemize}
\begin{itemize}
\item commit // A=100; B=50
\end{itemize}

\[\text{begin} // T2\]
\begin{itemize}
\item write(A, read(A)-20)
\item write(B, read(B)+20)
\end{itemize}
\begin{itemize}
\item commit // A=80; B=70
\end{itemize}

\[\text{begin} // T3\]
\begin{itemize}
\item write(A, read(A)+30)
\end{itemize}
\text{crash!}

\[\text{read(var):}\]
\begin{itemize}
\item if var in cache:
\item \hspace{1em} return cache[var]
\item else:
\item \hspace{2em} \begin{itemize}
\item // may evict others from cache to cell storage
\item cache[var] = cell_read(var)
\item return cache[var]
\end{itemize}
\end{itemize}

\[\text{write(var, value):}\]
\begin{itemize}
\item log.append(current_tid, update, var, read(var), value)
\item cache[var] = value
\end{itemize}

\[\text{flush():} \begin{itemize}
\item \hspace{1em} \text{called “occasionally”}
\item \hspace{2em} \text{cell_write(var, cache[var]) for each var}
\end{itemize}\]

\[\text{recover(log):}\]
\begin{itemize}
\item \hspace{1em} \text{commits = []}
\item \hspace{2em} \text{for record r in log[len(log)-1] .. log[0]:}
\item \hspace{3em} \begin{itemize}
\item \hspace{4em} \text{if r.type == COMMIT:}
\item \hspace{5em} \hspace{1em} \text{commits.add(r.tid)}
\item \hspace{4em} \text{if r.type == UPDATE and r.tid not in commits:}
\item \hspace{5em} \hspace{2em} \text{cell_write(r.var, r.old_val) // undo}
\end{itemize}
\end{itemize}

\[\text{commit = []}\]
read(var):
    if var in cache:
        return cache[var]
    else:
        // may evict others from cache to cell storage
        cache[var] = cell_read(var)
        return cache[var]

write(var, value):
    log.append(current_tid, update, var, read(var), value)
    cache[var] = value

flush(): // called “occasionally”
    cell_write(var, cache[var]) for each var

recover(log):
    commits = []
    for record r in log[len(log)-1] .. log[0]:
        if r.type == COMMIT:
            commits.add(r.tid)
        if r.type == UPDATE and r.tid not in commits:
            cell_write(r.var, r.old_val) // undo
    for record r in log[0] .. log[len(log)-1]:
        if r.type == UPDATE and r.tid in commits:
            cell_write(r.var, r.new_value) // redo

problem: recovery is still slow
solution: write checkpoints and truncate the log

begin // T1
write(A, 100)
write(B, 50)
commit // A=100; B=50

begin // T2
write(A, read(A)-20)
write(B, read(B)+20)
commit // A=80; B=70

begin // T3
write(A, read(A)+30)
crash!

read(var):
if var in cache:
    return cache[var]
else:
    // may evict others from cache to cell storage
    cache[var] = cell_read(var)
    return cache[var]

write(var, value):
log.append(current_tid, update, var, read(var), value)
cache[var] = value

flush(): // called “occasionally”
cell_write(var, cache[var]) for each var

recover(log):
    commits = []
    for record r in log[len(log)-1] .. log[0]:
        if r.type == COMMIT:
            commits.add(r.tid)
        if r.type == UPDATE and r.tid not in commits:
            cell_write(r.var, r.old_val) // undo
    for record r in log[0] .. log[len(log)-1]:
        if r.type == UPDATE and r.tid in commits:
            cell_write(r.var, r.new_value) // redo
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**: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity.

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

* shadow copies are used in some systems.
(write-ahead) logs provide atomicity with better performance than shadow copies. The primary benefit is making small appends for each update, rather than copy an entire file over for every change.

Cell storage is used with the log to improve read performance, and caches and truncation can be used to improve write and recovery performance.