superior to shadow copies in almost every way
Did One Guy Just Stop a Huge Cyberattack?

A Microsoft engineer noticed something was off on a piece of software he worked on. He soon discovered someone was probably trying to gain access to computers all over the world.
According to some researchers who have gone back and looked at the evidence, the attacker appears to have used a pseudonym, “Jia Tan,” to suggest changes to xz Utils as far back as 2022. (Many open-source software projects are governed via hierarchy; developers suggest changes to a program’s code, then more experienced developers known as “maintainers” have to review and approve the changes.)

The attacker, using the Jia Tan name, appears to have spent several years slowly gaining the trust of other xz Utils developers and getting more control over the project, eventually becoming a maintainer, and finally inserting the code with the hidden backdoor earlier this year. (The new, compromised version of the code had been released, but was not yet in widespread use.)

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.)
transfer \( (bank, \text{account}_a, \text{account}_b, \text{amount}) : \)
\[
\text{bank}[\text{account}_a] = \text{bank}[\text{account}_a] - \text{amount}
\]
\[
\text{bank}[\text{account}_b] = \text{bank}[\text{account}_b] + \text{amount}
\]

this was our starting bank transfer code from last week. let’s put it into transaction syntax.

\[
\begin{align*}
\text{begin} \\
\text{A} &= \text{read(A)} \\
\text{B} &= \text{read(B)} \\
\text{write}(\text{A}, \text{A}-\text{amount}) \\
\text{write}(\text{B}, \text{B}+\text{amount}) \\
\text{commit}
\end{align*}
\]

we can even be more succinct and get rid of the local variables

\[
\begin{align*}
\text{begin} \\
\text{write}(\text{A}, \text{read(A)}-\text{amount}) \\
\text{write}(\text{B}, \text{read(B)}+\text{amount}) \\
\text{commit}
\end{align*}
\]

to broaden our horizons, we’re going to move away from bank transfers and think about generic reads/writes
our goal today is to make sure that each transaction in a series (such as the one below) is **atomic and** that our system has good **performance**

remember that one problem with shadow copies is that they rewrite an entire file even for small changes

```plaintext
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!🌟
```
our goal today is to make sure that each transaction in a series (such as the one below) is **atomic and** that our system has good **performance**

remember that one problem with shadow copies is that they rewrite an entire file even for small changes

```plaintext
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)
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)
```

<table>
<thead>
<tr>
<th>TID</th>
<th>T1</th>
<th>T1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UPDATE</td>
<td>UPDATE</td>
<td>COMMIT</td>
</tr>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</td>
<td></td>
</tr>
<tr>
<td>NEW</td>
<td>A=100</td>
<td>B=50</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]
brief interlude: we’re going to change this example slightly, to illustrate one additional point

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

begin  // T2
write(A, read(A)-20)
write(A, read(A)-30)
commit  // A=50; B=50

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

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
Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

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! 

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

after a crash, the log is still correct; uncommitted updates will not be read
writes contain the old and new value of a variable. Each write is a small append to the end of the log.

to read a variable $x$, the system scans backwards through the log to find $x$'s last committed value.

the commit point for a transaction is writing the COMMIT record.

problem: reads can be very slow
Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

```
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></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)

read(var):
    return cell_read(var)

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

read(var):
    return cell_read(var)

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

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>50</td>
</tr>
</tbody>
</table>

### Cell Storage

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>80</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

### Read

```python
def read(var):
    return cell_read(var)
```

### Write

```python
def write(var, value):
    log.append(current_tid, "UPDATE", var, read(var), value)
    cell_write(var, value)
```

### Transactions

```python
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)```


```plaintext
// 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)
```

---

```
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>TID</th>
<th>T1 UPDATE</th>
<th>T1 COMMIT</th>
<th>T1 UPDATE</th>
<th>T2 UPDATE</th>
<th>T2 UPDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>A=0</td>
<td>B=0</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>A=80</td>
<td>B=70</td>
<td></td>
</tr>
</tbody>
</table>

---

**cell storage (on disk)**

```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>80</td>
<td>70</td>
</tr>
</tbody>
</table>
```

---

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

problem: the value of A in cell storage never committed (and so should not be read after recovery); 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)
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 = []
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

commits = [T2, T1]
**writes** go to the log first and then cell storage.

to **read** a variable $x$, the system reads $x$'s value from cell storage

on **recovery**, the system must repair cell storage by undo-ing any uncommitted transactions

**problem:** read performance is now great, but writes got slower
(recovery also got slower; we'll come to that)
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 slower
(recovery also got slower; we’ll come to that)
<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></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)
```
<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>

`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)

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

question: on a crash, could we have updates that should be in cell storage, but aren’t? what about changes that shouldn’t be in cell storage, but are?
### Transaction Log and Example

<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></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>

#### Example Transactions

- **Begin**: 
  - `A = 100` (T1)
  - `B = 50` (T1)
  - `commit`

- **Begin**: 
  - `A = read(A) - 20` (T2)
  - `B = read(B) + 20` (T2)
  - `commit`

- **Begin**: 
  - `A = read(A) + 30` (T3)
  - `crash!`

---

### Cache Management

- **Read**: 
  - `read(var)`: 
    - If `var` is in cache: return `cache[var]`
    - Else: 
      - May evict others from cache to cell storage
      - `cache[var] = cell_read(var)`
      - return `cache[var]`

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

- **Flush**: 
  - `flush()`: called “occasionally”
    - `cell_write(var, cache[var])` for each var

- **Recover**: 
  - `recover(log)`: 
    - `commits = []`
    - For each 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.**
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

    commits = []

all other updates were committed; B’s value won’t ever be changed
\begin{align*}
\textbf{problem:} & \quad \text{recovery is still slow} \\
\end{align*}
### Solution

Write checkpoints and truncate the log.
cell storage and the cache make reads and writes faster, but make our recovery process more complex. In particular, because cell storage is permanent, recovery must make sure it is correct — **undo**-ing any un-committed updates and **redo**-ing any updates that didn’t get flushed from the cache.

```python
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)
(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.

The addition of these performance-improving techniques makes the system’s recovery process more complex.