what do we want from isolation, and how do we get it?
To plan for an eclipse, electrical system operators need to figure out how much the energy production will drop and how much power people will draw from the reserves. On the day of the 2017 total solar eclipse, for example, solar power generation in the U.S. dropped 25% below average.

Because solar power production falls quickly during the eclipse’s peak, grid operators may need to tap into reserves at a rate that may strain the electrical transmission lines. To try to keep things running smoothly, grid operators will rely on local reserves and minimize power transfer between grids during the event. This should lessen the burden on transmission lines in local grids and prevent temporary blackouts.

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).
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:** provided by **two-phase locking**.

* shadow copies are used in some systems.
**goal:** run transactions $T_1$, $T_2$, $T_N$ concurrently, and have it “appear” as if they ran sequentially

what does this even mean?

when we run two transactions concurrently, we’ll always run the steps of a single transaction in order (e.g., $T_1.1$ before $T_1.2$). but we might interleave steps of $T_2$ in between steps of $T_1$.

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>$T1.1$</td>
<td>$T2.1$</td>
</tr>
<tr>
<td>read(x)</td>
<td>write(x, 20)</td>
</tr>
<tr>
<td>$T1.2$</td>
<td>$T2.2$</td>
</tr>
<tr>
<td>tmp = read(y)</td>
<td>write(y, 30)</td>
</tr>
<tr>
<td>$T1.3$</td>
<td></td>
</tr>
<tr>
<td>write(y, tmp+10)</td>
<td>commit</td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>
```

(assume x, y initialized to zero)

**naive approach:** actually run them sequentially, via (perhaps) a single global lock
**goal:** run transactions \( T_1, T_2 \) concurrently, and have it “appear” as if they ran sequentially

\[
\begin{align*}
\text{T1} & \text{ begin} \\
\text{T1.1} & \text{ read(x)} \\
\text{T1.2} & \text{ tmp = read(y)} \\
\text{T1.3} & \text{ write(y, tmp+10)} \quad \text{commit} \\
\end{align*}
\]

\[
\begin{align*}
\text{T2} & \text{ begin} \\
\text{T2.1} & \text{ write(x, 20)} \\
\text{T2.2} & \text{ write(y, 30)} \quad \text{commit} \\
\end{align*}
\]

(assume \( x, y \) initialized to zero)

**result:** \( x=20; y=30 \)

let’s look at a few different schedules of \( T_1 \) and \( T_2 \) (this is not an exhaustive list)

\[
\begin{align*}
\text{T2.1} & \text{ write(x, 20)} \\
\text{T1.1} & \text{ read(x)} \\
\text{T2.2} & \text{ write(y, 30)} \\
\text{T1.2} & \text{ tmp = read(y)} \\
\text{T1.3} & \text{ write(y, tmp+10)} \\
\text{result: x=20; y=40} \\
\end{align*}
\]

\[
\begin{align*}
\text{T1.1} & \text{ read(x)} \\
\text{T2.1} & \text{ write(x, 20)} \\
\text{T1.2} & \text{ tmp = read(y)} \\
\text{T2.2} & \text{ write(y, 30)} \\
\text{T1.3} & \text{ write(y, tmp+10)} \\
\text{result: x=20; y=10} \\
\end{align*}
\]

\[
\begin{align*}
\text{T1.1} & \text{ read(x)} \\
\text{T2.1} & \text{ write(x, 20)} \\
\text{T2.2} & \text{ write(y, 30)} \\
\text{T1.2} & \text{ tmp = read(y)} \\
\text{T1.3} & \text{ write(y, tmp+10)} \\
\text{result: x=20; y=40} \\
\end{align*}
\]
goal: run transactions T1, T2 concurrently, and have it “appear” as if they ran sequentially

T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit

T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit

(result: x=20; y=30)

let’s look at a few different schedules of T1 and T2 (this is not an exhaustive list)

T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)

result: x=20; y=40

T2.1 write(x, 20)
T2.2 write(y, 30)

result: x=20; y=40

result: x=20; y=10

(result: x=20; y=40)

it seems like the middle schedule is out; x=20; y=10 is not possible in either of our serialized schedules

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024
**goal:** run transactions **T1, T2** concurrently, and have it “appear” as if they ran sequentially

(assume x, y initialized to zero)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>begin</strong></td>
<td><strong>begin</strong></td>
</tr>
<tr>
<td><strong>T1.1</strong> read(x)</td>
<td><strong>T2.1</strong> write(x, 20)</td>
</tr>
<tr>
<td><strong>T1.2</strong> tmp = read(y)</td>
<td><strong>T2.1</strong> write(x, 20)</td>
</tr>
<tr>
<td><strong>T1.3</strong> write(y, tmp+10)</td>
<td><strong>T2.2</strong> write(y, 30)</td>
</tr>
<tr>
<td><strong>commit</strong></td>
<td><strong>commit</strong></td>
</tr>
</tbody>
</table>

result: x=20; y=40

**let’s look at a few different schedules of T1 and T2 (this is not an exhaustive list)**

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T2.1</strong> write(x, 20)</td>
<td><strong>T1.1</strong> read(x)</td>
<td><strong>T1.1</strong> read(x)</td>
</tr>
<tr>
<td><strong>T1.1</strong> read(x)</td>
<td><strong>T2.1</strong> write(x, 20)</td>
<td><strong>T2.1</strong> write(x, 20)</td>
</tr>
<tr>
<td><strong>T2.2</strong> write(y, 30)</td>
<td><strong>T1.2</strong> tmp = read(y)</td>
<td><strong>T1.2</strong> tmp = read(y)</td>
</tr>
<tr>
<td><strong>T1.2</strong> tmp = read(y)</td>
<td><strong>T2.2</strong> write(y, 30)</td>
<td><strong>T2.2</strong> write(y, 30)</td>
</tr>
<tr>
<td><strong>T1.3</strong> write(y, tmp+10)</td>
<td><strong>T1.3</strong> write(y, tmp+10)</td>
<td><strong>T1.3</strong> write(y, tmp+10)</td>
</tr>
<tr>
<td><strong>result:</strong> x=20; y=40</td>
<td><strong>result:</strong> x=20; y=10</td>
<td><strong>result:</strong> x=20; y=40</td>
</tr>
</tbody>
</table>

but take a closer look at the third schedule; in the first step, **T1.1** reads x=0, and in the fourth step, **T1.2** reads y=30. those two reads together aren’t possible in either sequential schedule. Is that okay?
it depends.

there are many ways for multiple transactions to “appear” to have been run in sequence; we say there are different notions of serializability. what type of serializability you want depends on what your application needs.
conflicts: two operations conflict if they operate on the same object and at least one of them is a write

![Diagram of two transactions T1 and T2]

in any schedule, two conflicting operations A and B will have an order: either A is executed before B, or B is executed before A. we'll call this the order of the conflict (in that schedule).

order of conflicts

T1.1 read(x) -> T2.1 write(x, 20)
T1.2 tmp = read(y) -> T2.2 write(y, 30)
T1.3 write(y, tmp+10) -> T2.2 write(y, 30)
**conflicts:** two operations conflict if they operate on the same object and at least one of them is a write

### T1

T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit

(assume x, y initialized to zero)

### T2

begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit

### conflicts

- T1.1 read(x) and T2.1 write(x, 20)
- T1.2 tmp = read(y) and T2.2 write(y, 30)
- T1.3 write(y, tmp+10) and T2.2 write(y, 30)

in any schedule, two conflicting operations A and B will have an order: either A is executed before B, or B is executed before A. we'll call this the **order** of the conflict (in that schedule).

| T1.1 read(x) | T1.2 tmp = read(y) | T1.3 write(y, tmp+10) | order of conflicts |
| T2.1 write(x, 20) | T2.2 write(y, 30) | | T1.1 -> T2.1 |
| T1.2 -> T2.2 |
| T1.3 -> T2.2 |

notice that, if we execute **T1** and **T2** serially, then in the ordering of the conflicts we see either *all* of **T1**'s operations occurring first, or *all* of **T2**'s operations occurring first

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024
conflicts: two operations conflict if they operate on the same object and at least one of them is a write

\[
\begin{align*}
T1 & \text{ begin} \\
T1.1 & \text{ read(x)} \\
T1.2 & \text{ tmp = read(y)} \\
T1.3 & \text{ write(y, tmp+10)} \\
\text{commit} \\
T2 & \text{ begin} \\
T2.1 & \text{ write(x, 20)} \\
T2.2 & \text{ write(y, 30)} \\
\text{commit}
\end{align*}
\]

(assume x, y initialized to zero)

in any schedule, two conflicting operations A and B will have an order: either A is executed before B, or B is executed before A. we’ll call this the order of the conflict (in that schedule).

\[
\begin{align*}
\text{order of conflicts} & \text{ T2.1 write(x, 20) and T2.1 write(x, 20)} \\
\text{order of conflicts} & \text{ T1.1 read(x) and T2.1 write(x, 20)} \\
\text{order of conflicts} & \text{ T1.2 tmp = read(y) and T2.2 write(y, 30)} \\
\text{order of conflicts} & \text{ T1.3 write(y, tmp+10) and T2.2 write(y, 30)}
\end{align*}
\]

on the left schedule, the order of conflicts is the same as if we had run T2 entirely before T1; on the right schedule, the order of conflicts isn’t the same as either serial schedule.
a schedule is **conflict serializable** if the order of all of its conflicts is the same as the order of the conflicts in some sequential schedule.

**T1**

begin

T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)

commit

**T2**

begin

T2.1 write(x, 20)
T2.2 write(y, 30)

commit

(assume x, y initialized to zero)

**order of conflicts**

T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3

this schedule is conflict serializable

**T1.1 read(x) and T2.1 write(x, 20)**
**T1.2 tmp = read(y) and T2.2 write(y, 30)**
**T1.3 write(y, tmp+10) and T2.2 write(y, 30)**

**conflicts**

**order of conflicts**

T1.1 -> T2.1
T2.2 -> T1.2
T2.2 -> T1.3

this schedule is **not** conflict serializable
A schedule is **conflict serializable** if the order of all of its conflicts is the same as the order of the conflicts in some sequential schedule.

**Example:**

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```

```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

(assume \(x, y\) initialized to zero)

We can express the order of conflicts more succinctly with a **conflict graph**: there is an edge from \(T_i\) to \(T_j\) if and only if \(T_i\) and \(T_j\) have a conflict between them and the first step in the conflict occurs in \(T_i\).

**Conflict Graph:**

```
T1.1 read(x) and T2.1 write(x, 20)
T1.2 tmp = read(y) and T2.2 write(y, 30)
T1.3 write(y, tmp+10) and T2.2 write(y, 30)
```

This schedule is conflict serializable.

```
T2.1 write(x, 20)
T1.1 read(x)
T2.2 write(y, 30)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
```

Conflict Graph:

```
T2 → T1
```

This schedule is **not** conflict serializable.
interlude: practice with \textit{conflict graphs}

\begin{itemize}
\item \textbf{T1} \begin{itemize}
\item \textbf{T1.1} read(x)
\item \textbf{T1.2} write(y, 10)
\end{itemize}
\item \textbf{T2} \begin{itemize}
\item \textbf{T2.1} write(x, 20)
\item \textbf{T2.2} write(y, 30)
\end{itemize}
\item \textbf{T3} \begin{itemize}
\item \textbf{T3.1} read(y)
\item \textbf{T3.2} write(z, 40)
\end{itemize}
\item \textbf{T4} \begin{itemize}
\item \textbf{T4.1} read(y)
\end{itemize}
\end{itemize}

\textbf{T1.1} read(x)\\% T2.1 write(x, 20)\% T3.1 read(y)\% T4.1 read(y)\% T1.2 write(y, 10)\% T2.2 write(y, 30)\% T3.2 write(z, 40)

\textbf{what is the conflict graph for this schedule?}
A schedule is **conflict serializable** if the order of all of its conflicts is the same as the order of the conflicts in some sequential schedule.

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit

T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

(assume x, y initialized to zero)

**conflicts**

- T1.1 read(x) and T2.1 write(x, 20)
- T1.2 tmp = read(y) and T2.2 write(y, 30)
- T1.3 write(y, tmp+10) and T2.2 write(y, 30)

We can express the order of conflicts more succinctly with a **conflict graph**: there is an edge from $T_i$ to $T_j$ if and only if $T_i$ and $T_j$ have a conflict between them and the first step in the conflict occurs in $T_i$.

```
T2.1 write(x, 20)
T1.1 read(x)
T2.2 write(y, 30)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
```

**conflict graph**

```
T2 ─→ T1
```

This schedule is conflict serializable.

```
T1.1 read(x)
T2.1 write(x, 20)
T2.2 write(y, 30)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
```

**conflict graph**

```
T2 ⇐ T1
```

This schedule is **not** conflict serializable.

A schedule is conflict serializable if and only if it has an acyclic conflict graph.
a schedule is conflict serializable if the order of all of its conflicts is the same as the order of the conflicts in some sequential schedule.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>T1.1 read(x)</td>
<td>T2.1 write(x, 20)</td>
</tr>
<tr>
<td>T1.2 tmp = read(y)</td>
<td>T2.2 write(y, 30)</td>
</tr>
<tr>
<td>T1.3 write(y, tmp+10)</td>
<td>commit</td>
</tr>
</tbody>
</table>

(assume x, y initialized to zero)

our goal (in lecture) is to run transactions concurrently, but to produce a schedule that is conflict serializable

how does a system do that? one way might be to generate all possible schedules and check their conflict graphs, and run one of the schedules with an acyclic conflict graph, but this will take some time
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

2PL still gives us options for where we place the locks

```
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
```
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

2PL still gives us options for where we place the locks

T1
begin
acquire(x.lock)
acquire(y.lock)
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
release(x.lock)
release(y.lock)
commit
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

2PL still gives us options for where we place the locks

T1
begin
acquire(x.lock)
T1.1 read(x)
acquire(y.lock)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
release(x.lock)
release(y.lock)
commit
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

2PL still gives us options for where we place the locks

```
T1
begin
acquire(x.lock)
T1.1 read(x)
acquire(y.lock)
release(x.lock)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
release(y.lock)
commit
```
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

2PL still gives us options for where we place the locks

T1
begin
acquire(x.lock)
T1.1 read(x)
release(x.lock)
acquire(y.lock)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
release(y.lock)
commit

we can't do this; it breaks the third rule of 2PL
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

if we release locks after commit, that is technically strict two-phase locking

2PL still gives us options for where we place the locks

```
T1
begin
acquire(x.lock)
T1.1 read(x)
acquire(y.lock)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
release(x.lock)
release(y.lock)
```

```
T2
begin
acquire(x.lock)
T2.1 write(x, 20)
acquire(y.lock)
T2.2 write(y, 30)
commit
release(x.lock)
release(y.lock)
```

notice that with this approach to 2PL, we will effectively force these two transactions to run serially. we’ll address that in a few slides!

there are some lingering issues related to possible deadlocks and performance; we’ll deal with those, but let’s first try to understand why 2PL produces a conflict-serializable schedule
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

2PL produces a conflict-serializable schedule (equivalently, 2PL produces a conflict graph without a cycle)

proof: suppose not. then a cycle exists in the conflict graph

\[ T_1 \xrightarrow{x_1} T_2 \xrightarrow{x_2} T_3 \xrightarrow{x_3} \ldots \xrightarrow{x_{k-1}} T_k \]

to cause the conflict, each pair of conflicting transactions must have some shared variable that they conflict on

\[ T_1 \text{ acquires } x_1.\text{lock} \]
\[ T_2 \text{ acquires } x_1.\text{lock} \]
\[ T_2 \text{ acquires } x_2.\text{lock} \]
\[ T_3 \text{ acquires } x_2.\text{lock} \]
\[ \ldots \]
\[ T_k \text{ acquires } x_k.\text{lock} \]
\[ T_1 \text{ acquires } x_k.\text{lock} \]

in the schedule, each pair of transactions needs to acquire a lock on their shared variable

the order of the conflict tells us which transaction acquired the lock first
two-phase locking (2PL)

1. each shared variable has a lock
2. before any operation on a variable, the transaction must acquire the corresponding lock
3. after a transaction releases a lock, it may not acquire any other locks

2PL produces a conflict-serializable schedule (equivalently, 2PL produces a conflict graph without a cycle)

**proof:** suppose not. then a cycle exists in the conflict graph

\[ T_1 \xrightarrow{x_1} T_2 \xrightarrow{x_2} T_3 \xrightarrow{x_3} \ldots \xrightarrow{x_{k-1}} T_k \]

- to cause the conflict, each pair of conflicting transactions must have some shared variable that they conflict on
- in the schedule, each pair of transactions needs to acquire a lock on their shared variable
- the order of the conflict tells us which transaction acquired the lock first
- in order for the schedule to progress, \( T_1 \) must have released its lock on \( x_1 \) before \( T_2 \) acquired it
two-phase locking (2PL)

1. each shared variable has a lock
2. before any operation on a variable, the transaction must acquire the corresponding lock
3. after a transaction releases a lock, it may not acquire any other locks

2PL produces a conflict-serializable schedule
(equivalently, 2PL produces a conflict graph without a cycle)

**proof:** suppose not. then a cycle exists in the conflict graph

to cause the conflict, each pair of conflicting transactions must have some shared variable that they conflict on

in the schedule, each pair of transactions needs to acquire a lock on their shared variable

the order of the conflict tells us which transaction acquired the lock first

in order for the schedule to progress, \( T_1 \) must have released its lock on \( x_1 \) before \( T_2 \) acquired it

**contradiction:** this is not a valid 2PL schedule
two-phase locking (2PL)

1. each shared variable has a lock
2. before any operation on a variable, the transaction must acquire the corresponding lock
3. after a transaction releases a lock, it may not acquire any other locks

problem: 2PL can result in deadlock

T1
begin
acquire(x.lock)
T1.1 read(x)
acquire(y.lock)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
release(x.lock)
release(y.lock)

T2
begin
acquire(x.lock)
T2.1 write(x, 20)
acquire(y.lock)
T2.2 write(y, 30)
commit
release(x.lock)
release(y.lock)
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

problem: 2PL can result in deadlock

T1
begin
acquire(x.lock)
T1.1 read(x)
acquire(y.lock)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
release(x.lock)
release(y.lock)

T2
begin
acquire(y.lock)
T2.1 write(y, 30)
acquire(x.lock)
T2.2 write(x, 20)
commit
release(x.lock)
release(y.lock)

for example, suppose T2 wrote to y before x

one solution to this problem is a global ordering on locks; but we hate that! a better solution is to take advantage of atomicity and abort one of the transactions
two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks

problem: performance

T1
begin
acquire(x.lock)
T1.1 read(x)
acquire(y.lock)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit
release(x.lock)
release(y.lock)

T2
begin
acquire(x.lock)
T2.1 write(x, 20)
acquire(y.lock)
T2.2 write(y, 30)
commit
release(x.lock)
release(y.lock)
two-phase locking (2PL)
with reader-/writer- locks

1. each shared variable has two locks: one for **reading**, one for **writing**

2. before **any** operation on a variable, the transaction must acquire the appropriate lock

3. multiple transactions can hold **reader** locks for the same variable at once; a transaction can only hold a **writer** lock for a variable if there are **no** other locks held for that variable

4. after a transaction releases a lock, it may **not** acquire any other locks

---

**problem:** performance

```
T1
begin
acquire(x.reader_lock)  
T1.1 read(x) 
acquire(y.reader_lock)  
T1.2 tmp = read(y) 
acquire(y.writer_lock)  
T1.3 write(y, tmp+10) 
commit
release(x.reader_lock) 
release(y.reader_lock)  
release(y.writer_lock)

T2
begin
acquire(x.writer_lock)  
T2.1 write(x, 20) 
acquire(y.writer_lock)  
T2.2 write(y, 30) 
commit
release(x.writer_lock) 
release(y.writer_lock)
```

we will often release reader locks before the commit
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**: provided by **two-phase locking**

* shadow copies are used in some systems
different types of **serializability** allow us to specify precise what we want when we run transactions in parallel. **conflict-serializability** is a relatively strict form of serializability.

**two-phase locking** allows us to generate conflict-serializable schedules. we can improve its performance by allowing concurrent reads via reader- and writer- locks.

2PL does not produce every possible conflict-serializable schedule — that’s okay! the claim is only that the schedules it does produce are conflict-serializable.