6.033 in the news

But that kind of invasive tracking is being scaled back or blocked by Apple and Google to protect people’s privacy. Last April, Apple introduced a feature so iPhones users could choose not to be followed by different apps. Google also announced a plan to disable the tracking tech in its Chrome web browser by 2023 and said it was working to limit data sharing on Android phones.

Now tracking has shifted to what is known as “first party” tracking. With this method, people are not being trailed from app to app or site to site. But companies are still gathering information on what people are doing on their specific site or app, with users' consent. This kind of tracking, which companies have practiced for years, is growing.

https://www.nytimes.com/2022/04/06/technology/online-tracking-privacy.html
The rise of this tracking has implications for digital advertising, which has depended on user data to know where to aim promotions. It tilts the playing field toward large digital ecosystems such as Google, Snap, TikTok, Amazon and Pinterest, which have millions of their own users and have amassed information on them. Smaller brands have to turn to those platforms if they want to advertise to find new customers.

Many small businesses already appear to be spending less on digital ads that rely on third-party data, such as Facebook and Instagram ads, and are reallocating marketing budgets to platforms with lots of first-party information, like Google and Amazon.

why should “systems people” care about this issue?

https://www.nytimes.com/2022/04/06/technology/online-tracking-privacy.html
6.033 Spring 2022

Lecture #18: Isolation

what do we want from isolation, and how do we get it?
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
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 $T_1$, $T_2$, $\ldots$, $T_N$ concurrently, and have it “appear” as if they ran sequentially

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>begin</td>
</tr>
<tr>
<td>$T_{1.1}$ read(x)</td>
<td>$T_{2.1}$ write(x, 20)</td>
</tr>
<tr>
<td>$T_{1.2}$ tmp = read(y)</td>
<td>$T_{2.2}$ write(y, 30)</td>
</tr>
<tr>
<td>$T_{1.3}$ write(y, tmp+10)</td>
<td>commit</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>

(assume x, y initialized to zero)

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 $T_1$.

**naive approach:** actually run them sequentially, via (perhaps) a single global lock
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

(assume x, y initialized to zero)

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

result: x=20; y=30

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

T2.1 write(x, 20)
T2.2 write(y, 30)
result: x=20; y=40

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

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)
result: x=20; y=40

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)
result: x=20; y=10

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)
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
**goal:** run transactions **T1, T2** concurrently, and have it “appear” as if they ran sequentially

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

<table>
<thead>
<tr>
<th><strong>T2</strong></th>
<th><strong>T1</strong></th>
<th><strong>Result</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>read(x)</td>
<td>x=20; y=30</td>
</tr>
<tr>
<td>T2.1</td>
<td>write(x, 20)</td>
<td>x=20; y=40</td>
</tr>
<tr>
<td>T2.2</td>
<td>write(y, 30)</td>
<td></td>
</tr>
<tr>
<td>T2.1</td>
<td>tmp = read(y)</td>
<td></td>
</tr>
<tr>
<td>T2.2</td>
<td>write(y, 30)</td>
<td></td>
</tr>
<tr>
<td>result: x=20; y=40</td>
<td>result: x=20; y=10</td>
<td>result: 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 a sequential schedule. **Is that okay?**
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

```
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: two operations conflict if they operate on the same object and at least one of them is a write.

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

order of conflicts

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
**conflicts:** two operations conflict if they operate on the same object and at least one of them is a write

<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>
<tr>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

(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).

<table>
<thead>
<tr>
<th>order of conflicts</th>
<th>T2.1 write(x, 20)</th>
<th>T1.1 read(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2.1 -&gt; T1.1</td>
<td>T2.2 -&gt; T1.2</td>
<td></td>
</tr>
<tr>
<td>T2.2 -&gt; T1.3</td>
<td>T1.2 tmp = read(y)</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>order of conflicts</th>
<th>T1.1 read(x)</th>
<th>T2.1 write(x, 20)</th>
<th>T2.2 write(y, 30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1.1 -&gt; T2.1</td>
<td>T2.2 -&gt; T1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2.2 -&gt; T1.3</td>
<td>T1.2 tmp = read(y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1.3 write(y, tmp+10)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

```plaintext
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)

The schedule:

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

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_1 to T_j if and only if T_1 and T_j have a conflict between them and the first step in the conflict occurs in T_1.

```plaintext
conflict graph
T2 → T1
```

This schedule is conflict serializable.

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

(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) |

**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
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

often we release locks after commit, which 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

\[
\begin{align*}
T_1 & \rightarrow x_1 \quad T_2 \rightarrow x_2 \quad T_3 \rightarrow x_3 \quad \ldots \rightarrow T_{k-1} \rightarrow T_k \\
& \quad x_k
\end{align*}
\]

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

\[
\begin{align*}
T_1 & \text{ acquires } x_1.\text{lock} \\
T_1 & \text{ releases } x_1.\text{lock} \\
T_2 & \text{ acquires } x_1.\text{lock} \\
T_2 & \text{ acquires } x_2.\text{lock} \\
T_3 & \text{ acquires } x_2.\text{lock} \\
& \quad \ldots
\end{align*}
\]

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

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.

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.

```
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.
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

\[
\text{begin} \\
\text{acquire}(x.\text{reader_lock}) \\
\text{acquire}(y.\text{reader_lock}) \\
\text{end}
\]

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

\[
\text{commit}
\]

T2

\[
\text{begin} \\
\text{acquire}(x.\text{writer_lock}) \\
\text{acquire}(y.\text{writer_lock}) \\
\text{end}
\]

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

\[
\text{commit}
\]

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