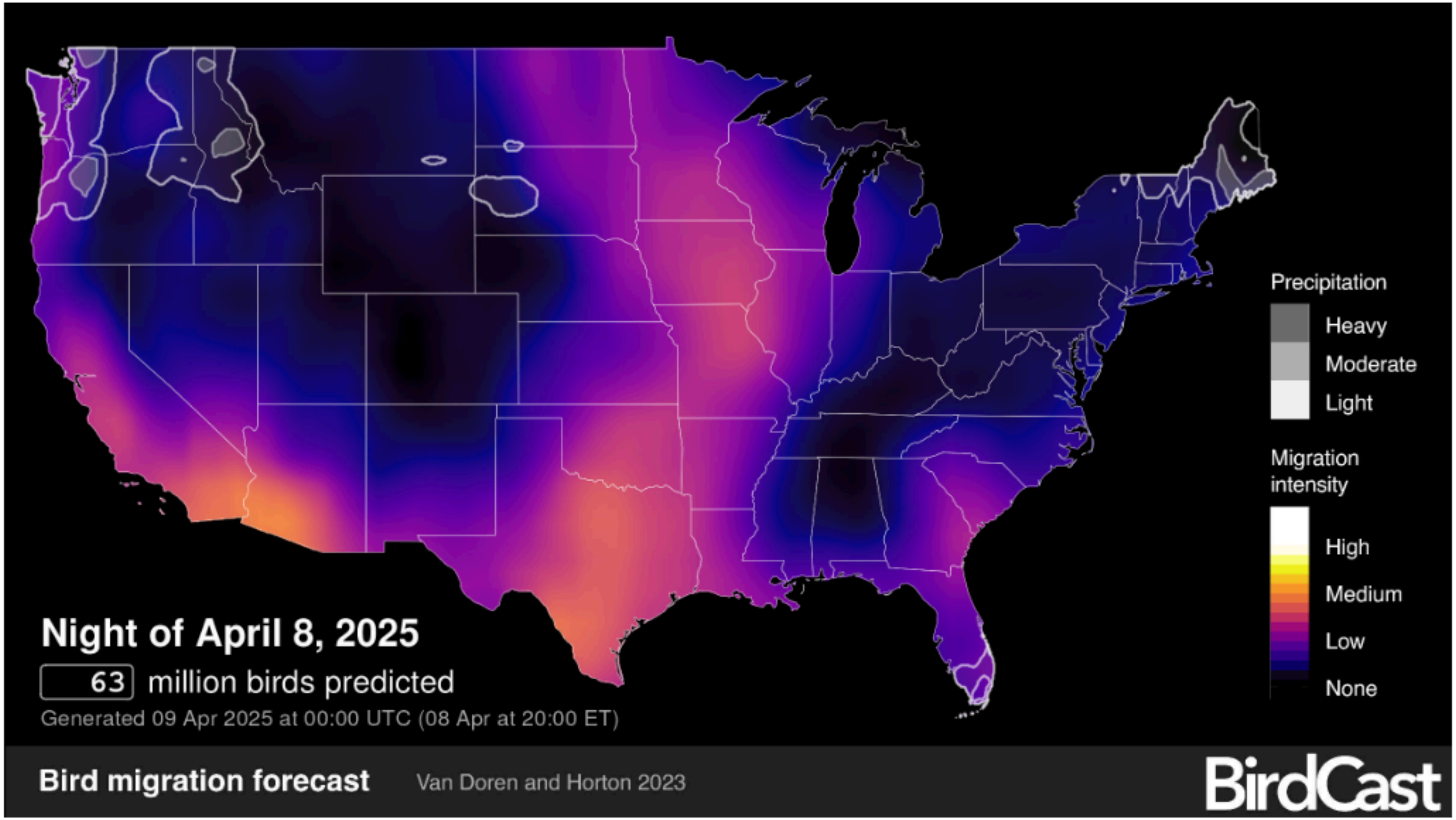
6.1800 Spring 2025

Lecture #18: Isolation

what do we want from isolation, and how do we get it?

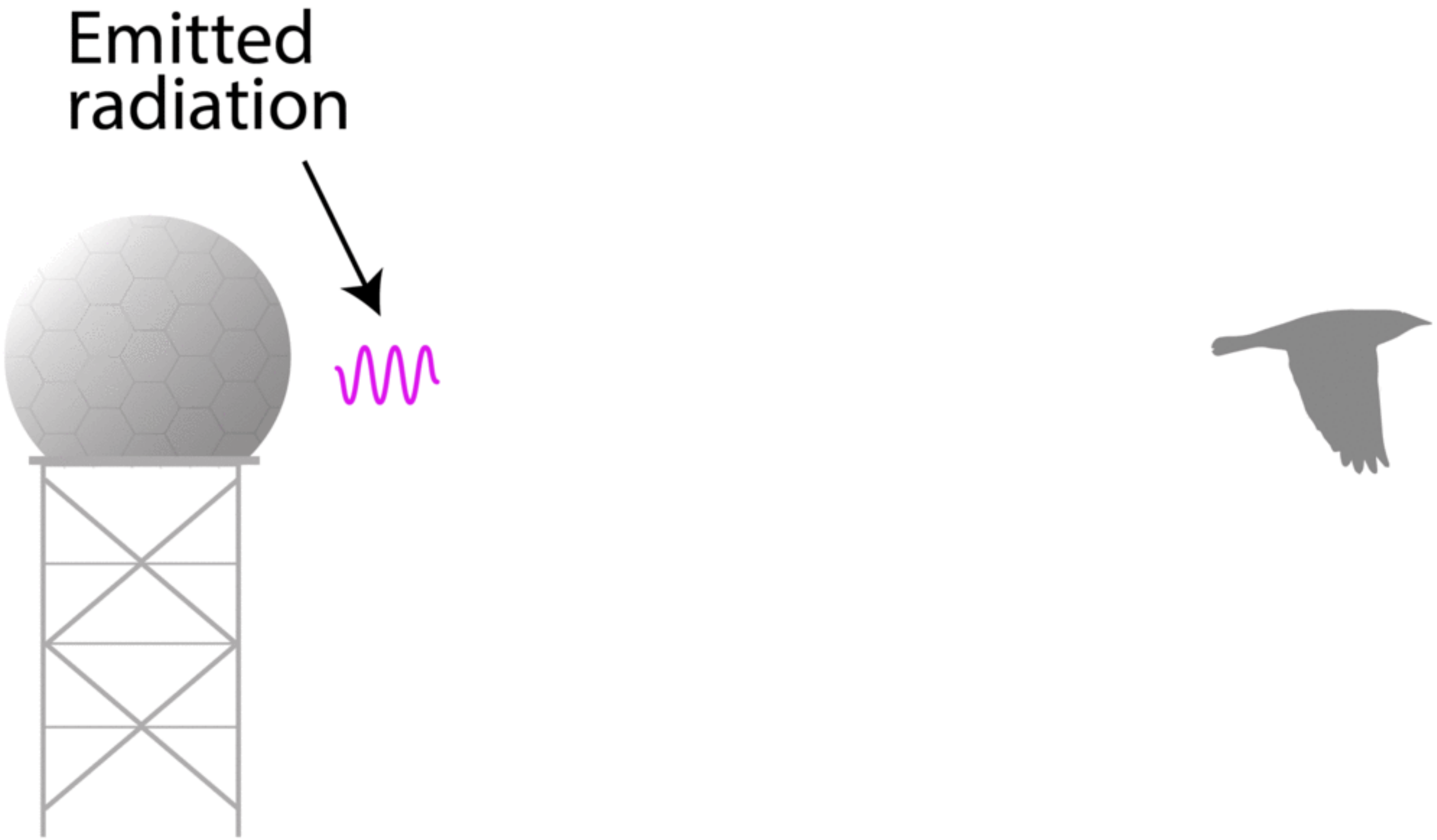
Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

6.1800 in the news



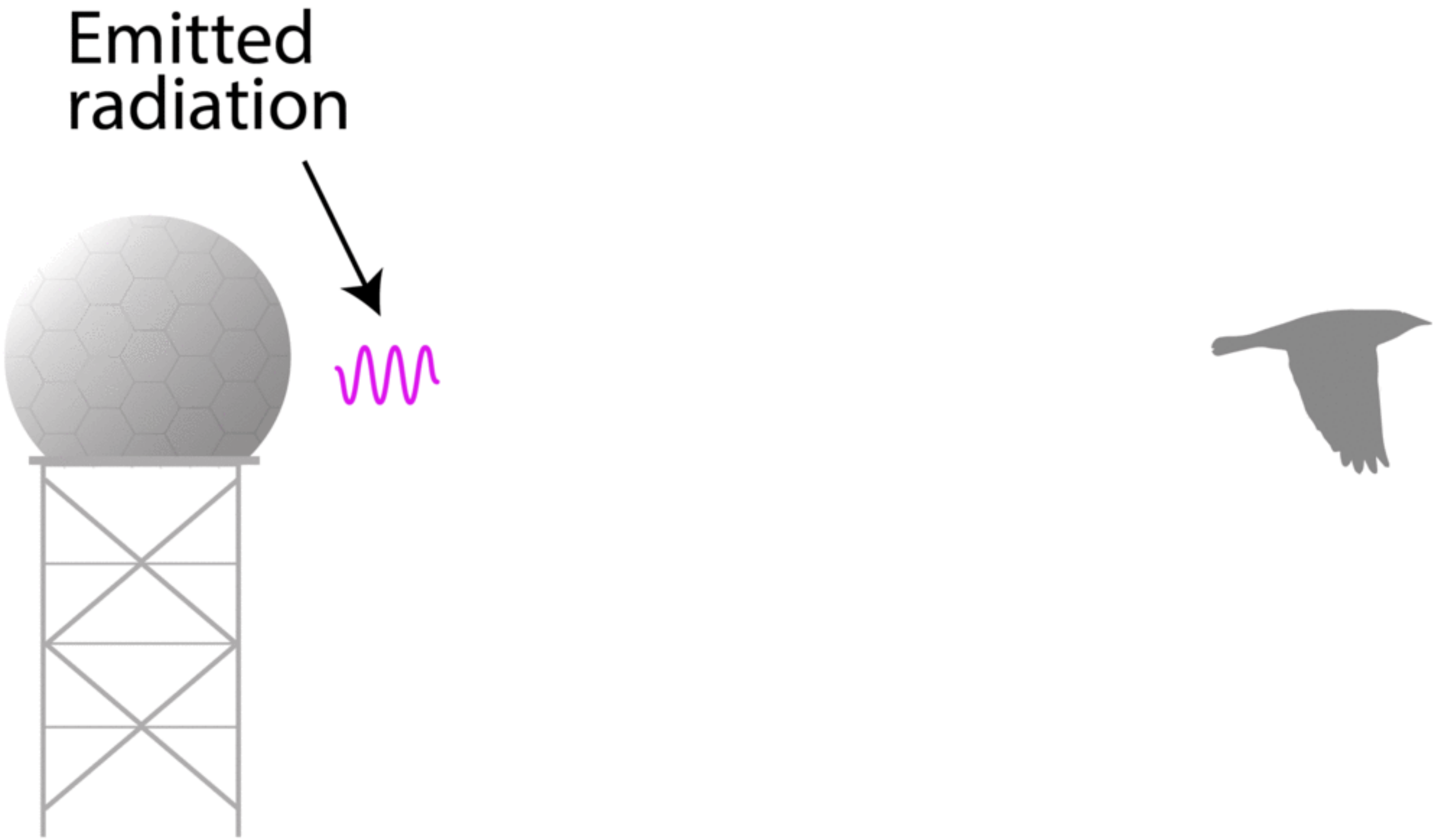
https://birdcast.info/migration-tools/migration-forecast-maps/

6.1800 in the news



https://birdcast.info/news/frequently-asked-questions/

6.1800 in the news



https://birdcast.info/news/frequently-asked-questions/

6.1800 in the news

A DISTRIBUTED ARCHITECTURE FOR THE WSR-88D (NEXRAD) RADAR PRODUCT GENERATOR (RPG)

Allen Zahrai[1], Zhongqi Jing[1,2], and Neil Peery[3]

[1]NOAA/ERL/National Severe Storms Laboratory,
Norman, Oklahoma

[2]Cooperative Institute for Mesoscale Meteorological Studies,
University of Oklahoma, Norman, Oklahoma

[3]NOAA/NWS/WSR-88D Operational Support Facility,
Norman, Oklahoma

Architectural limitations of the existing system result from a design which is rigidly tied to a legacy computing platform. In this design, a single computer often performs many computationally dissimilar functions. For example, within the RPG functional area, a single computer performs large number of compute-intensive algorithms as well as extensive amount of input/output (I/O) operations.

Long-term viability of the WSR-88D system mandates transition to an open system architecture. Open system standards are international standards for hardware and software where products and modules from different vendors can interoperate. These standards were established to make modular replacement and incremental upgrades relatively simple and cost effective.

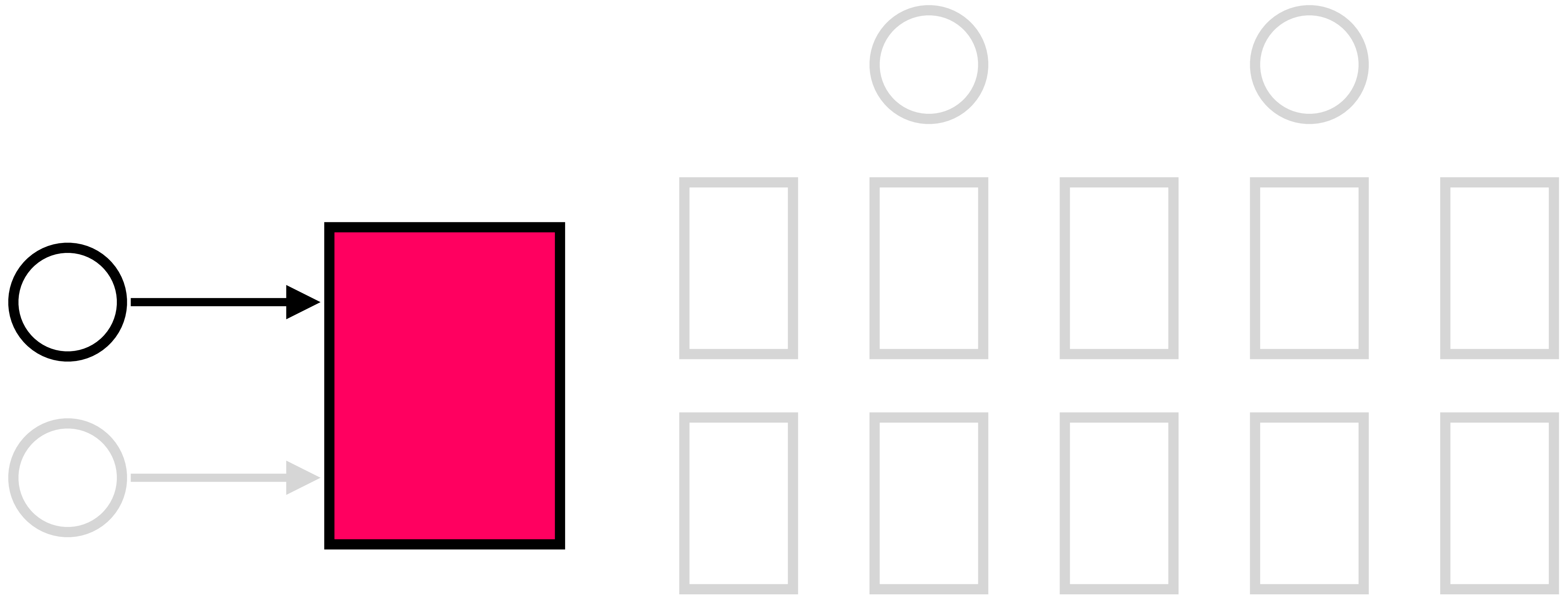
6.1800 in the news

Redundancy For Commercial T1 Service Between Remote Radars and NWS WFOs:

Backup communications systems to remote radars have increased the remote radar data availability and have mitigated loss of radar data over existing aged terrestrial circuits, especially during and after significant weather events. Remote NEXRAD NWS, FAA and DoD radars are geographically separated from their hosting National Weather Service (NWS) Weather Forecast Offices (WFOs). The current backup communications methods are cellular 4G Long Term Evolution (LTE) technology, and satellite Very Small Aperture Terminals (VSATs) at sites that cannot achieve adequate 4G LTE signal strength. The NEXRAD router configurations and auto fail-over mechanisms for 4G backup systems are the same as for VSAT backup systems, such that either 4G or VSAT technology may be deployed to any given site, depending on service capabilities and cost efficiencies. Both backup technologies have been successfully installed and are currently in operation across the NEXRAD fleet. The EHB 6-540 has been updated with preventative maintenance instructions and test procedures for NEXRAD field technicians to maintain the backup communication systems. Four portable 4G backup systems and four auto-deploy VSAT systems (fly-aways) are maintained by Systems Engineering. These systems have been deployed several times to support site-specific engineering solutions for radar data recovery, and to establish temporary communications paths for radars when required. Please contact the NEXRAD Hotline for specialized engineering support and deployable backup communication hardware requests.

https://www.roc.noaa.gov/branches/engineering-branch/systems-engineering.php

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

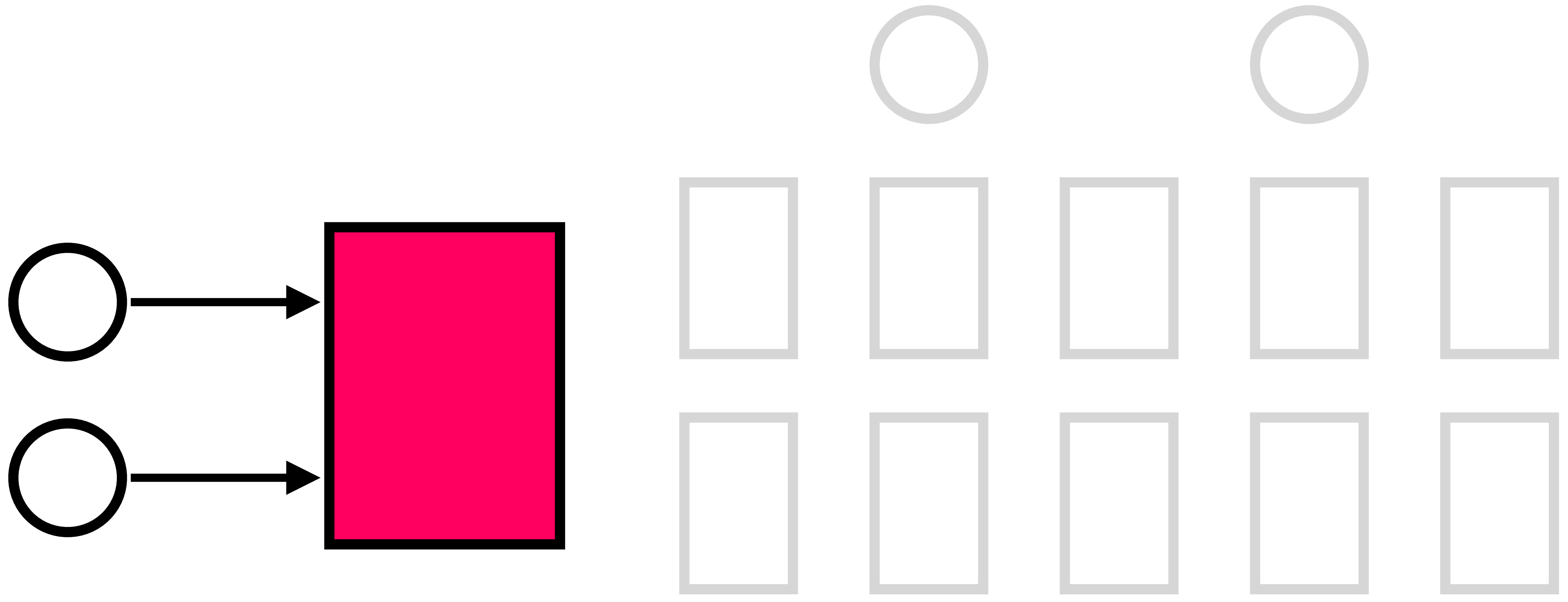
our job in lecture is to understand how a system *implements* these two abstractions. how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity

* shadow copies *are* used in some systems

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

* shadow copies *are* used in some systems

isolation: provided by **two-phase locking**

goal: run transactions `T1`, `T2`, .., `TN` concurrently, and have it "appear" as if they ran sequentially

goal: run transactions `T1`, `T2`, .., `TN` 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
```

(assume x, y initialized to zero)

goal: run transactions `T1`, `T2`, .., `TN` 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
```

(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., `T1.1` before `T1.2`). but we might interleave steps of `T2` in between steps of `T1`.

goal: run transactions `T1`, `T2`, .., `TN` concurrently, and have it "appear" as if they ran sequentially

```
T1                              T2
begin                           begin
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)           commit
commit
```

(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., `T1.1` before `T1.2`). but we might interleave steps of `T2` in between steps of `T1`.

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

goal: run transactions `T1`, `T2`, .., `TN` concurrently, and have it “appear” as if they ran sequentially

what does this even mean?

```
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)

when we run two transactions concurrently, we'll always run the steps of a single transaction in order (e.g., `T1.1` before `T1.2`). but we might interleave steps of `T2` in between steps of `T1`.

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
```

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

(assume x, y initialized to zero)

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
```

(assume x, y initialized to zero)

```
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)
```

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

result: x=20; y=40
```

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

result: x=20; y=40
```

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

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

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)
```

goal: run transactions T1, T2 concurrently, and have it "appear" as if they ran sequentially

```
T1                            T2
begin                         begin
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)         commit
commit
```

(assume x, y initialized to zero)

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

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

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)
```

```
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)
```

```
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)
```

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

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)
```

```
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)
```

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

```
T1                              T2
begin                           begin
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)           commit
commit
```

(assume x, y initialized to zero)

```
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)           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=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)
```

goal: run transactions T1, T2 concurrently, and have it "appear" as if they ran sequentially

```
T1                              T2
begin                           begin
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)           commit
commit
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

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
```

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

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

```
T1                               T2
begin                            begin
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)            commit
commit
```

(assume x, y initialized to zero)

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

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

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
```

(assume x, y initialized to zero)

```
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)

result: x=20; y=30
```

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

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) // x=0
T2.1 write(x, 20)
T2.2 write(y, 30)
T1.2 tmp = read(y) // y=30
T1.3 write(y, tmp+10)

result: x=20; y=40
```

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**.

goal: run transactions T1, T2 concurrently, and have it "appear" as if they ran sequentially

```
T1                                T2
begin                             begin
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)             commit
commit
```

(assume x, y initialized to zero)

```
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)             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=30                result: x=20; y=40
T1 reads x=0; y=0                 T1 reads x=20; y=30
```

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)               T1.1 read(x) // x=0
T1.1 read(x)               T2.1 write(x, 20)          T2.1 write(x, 20)
T2.2 write(y, 30)          T1.2 tmp = read(y)         T2.2 write(y, 30)
T1.2 tmp = read(y)         T2.2 write(y, 30)          T1.2 tmp = read(y) // y=30
T1.3 write(y, tmp+10)      T1.3 write(y, tmp+10)      T1.3 write(y, tmp+10)

result: x=20; y=40         result: x=20; y=10         result: x=20; y=40
```

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**.

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

```
T1                          T2
begin                       begin
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)       commit
commit
```

(assume x, y initialized to zero)

```
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)       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=30          result: x=20; y=40
T1 reads x=0; y=0           T1 reads x=20; y=30
```

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) // x=0
T2.1 write(x, 20)
T2.2 write(y, 30)
T1.2 tmp = read(y) // y=30
T1.3 write(y, tmp+10)

result: x=20; y=40
```

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

```
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

```
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

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

`T1.1 read(x)` and `T2.1 write(x, 20)`

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

`T1.1 read(x)` and `T2.1 write(x, 20)`

`T1.2 tmp = read(y)` and `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
```

```
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)`

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

`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).

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

```
T1                                T2
begin                             begin
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)             commit
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)`

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)
```

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

```
T1                              T2
begin                           begin
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)           commit
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)`

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 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
```

```
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)`

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
```

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

```
T1                               T2
begin                            begin
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)            commit
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)`

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
```

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

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

```
T1                                T2
begin                             begin
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)             commit
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)`

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
```

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

order of conflicts

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

`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
```

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

order of conflicts

```
T2.1 -> T1.1
```

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

```
T1                              T2
begin                           begin
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)           commit
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)`

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
```

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

order of conflicts

```
T2.1 -> T1.1
T2.2 -> T1.2
```

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

`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
```

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

order of conflicts

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

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

`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
```

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

order of conflicts

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

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

```
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)
```

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).

```
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)
```

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

```
T1                                  T2
begin                               begin
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)               commit
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)
```

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).

```
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)
```

order of conflicts

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

`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).

```
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)
```

order of conflicts

`T2.1 -> T1.1`

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

`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).

```
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)
```

order of conflicts

```
T2.1 -> T1.1
T2.2 -> T1.2
```

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

```
T1                          T2
begin                       begin
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)       commit
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)
```

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).

```
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)
```

order of conflicts

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

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

```
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).

```
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)
```

order of conflicts

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

```
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)
```

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

```
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).

```
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)
```

order of conflicts

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

```
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)
```

order of conflicts

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

```
T1                                  T2
begin                               begin
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)               commit
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)`

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).

```
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)
```

order of conflicts

`T2.1 -> T1.1`

`T2.2 -> T1.2`

`T2.2 -> T1.3`

```
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)
```

order of conflicts

`T1.1 -> T2.1`

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

```
T1                                  T2
begin                               begin
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)               commit
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)
```

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).

```
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)
```

order of conflicts

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

```
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)
```

order of conflicts

```
T1.1 -> T2.1
T2.2 -> T1.2
```

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

```
T1                                T2
begin                             begin
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)             commit
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)`

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).

```
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)
```

order of conflicts

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

```
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)
```

order of conflicts

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

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

```
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).

```
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)
```

order of conflicts

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

```
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)
```

order of conflicts

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

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)

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)
```

```
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)
```

order of conflicts

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

```
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)
```

order of conflicts

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

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)
```

```
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)
```

order of conflicts

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

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)
```

order of conflicts

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

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)
```

```
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)
```

order of conflicts

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

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)
```

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.

```
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 $\mathbf{T_i}$ to $\mathbf{T_j}$ if and only if $\mathbf{T_i}$ and $\mathbf{T_j}$ have a conflict between them and the first step in the conflict occurs in $\mathbf{T_i}$

```
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)
```

order of conflicts

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

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)
```

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.

```
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 $\longrightarrow$ 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 $\rightleftarrows$ T1

this schedule is **not** conflict serializable

interlude: practice with **conflict graphs**

```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

interlude: practice with **conflict graphs**

```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```

interlude: practice with **conflict graphs**

```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```

what is the conflict graph for this schedule?

interlude: practice with **conflict graphs**

```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```

T1 **T2** **T3** **T4**

what is the conflict graph for this schedule?

interlude: practice with **conflict graphs**

```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```

$T1 \longrightarrow T2 \quad T3 \quad T4$

what is the conflict graph for this schedule?

interlude: practice with **conflict graphs**

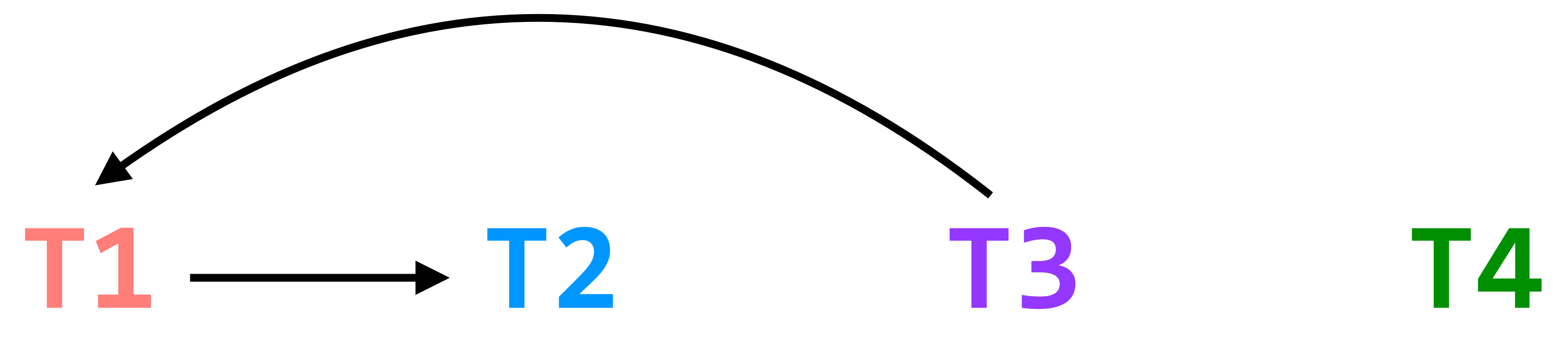
```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```



what is the conflict graph for this schedule?

interlude: practice with **conflict graphs**

```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

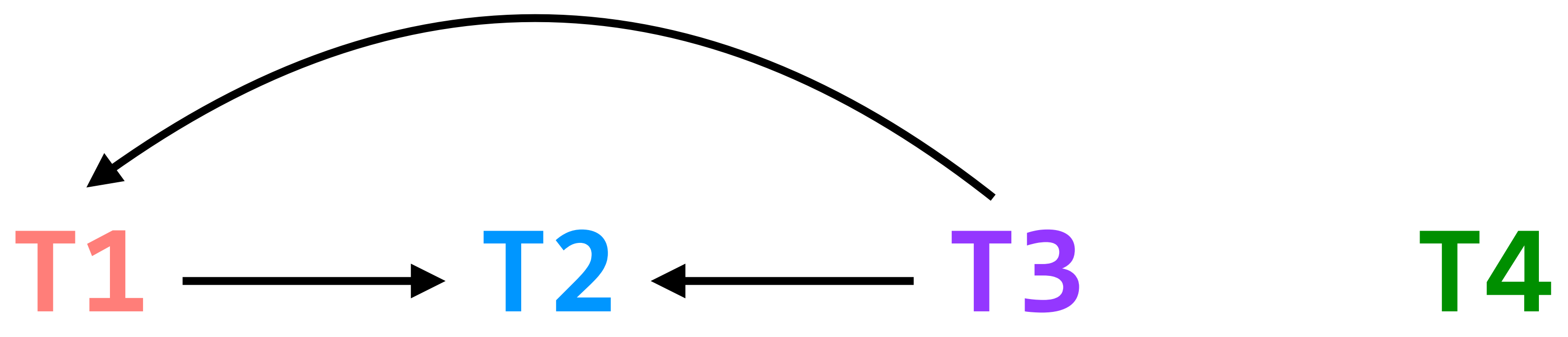
```
T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit
```

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```

what is the conflict graph for this schedule?



interlude: practice with **conflict graphs**

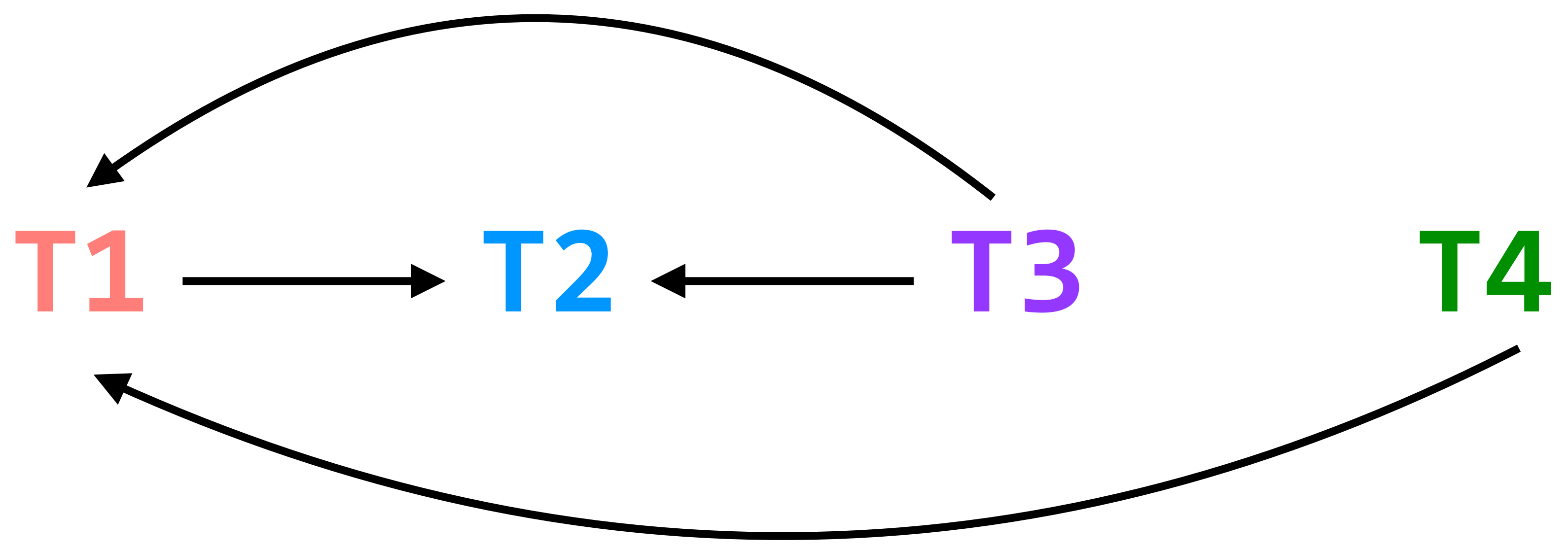
```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```



what is the conflict graph for this schedule?

interlude: practice with **conflict graphs**

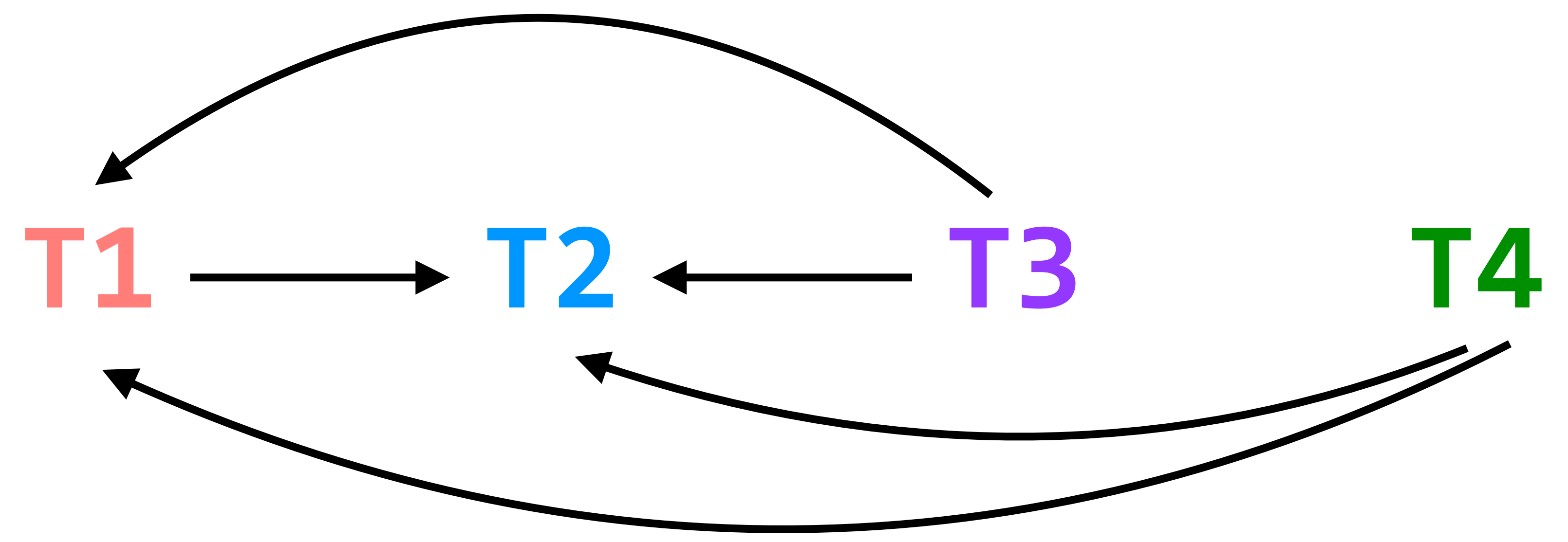
```
T1
begin
T1.1 read(x)
T1.2 write(y, 10)
commit
```

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

```
T3
begin
T3.1 read(y)
T3.2 write(z, 40)
commit
```

```
T4
begin
T4.1 read(y)
commit
```

```
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)
```



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 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 $\mathtt{T_i}$ to $\mathtt{T_j}$ if and only if $\mathtt{T_i}$ and $\mathtt{T_j}$ have a conflict between them and the first step in the conflict occurs in $\mathtt{T_i}$

```
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

$\mathtt{T2} \longrightarrow \mathtt{T1}$

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

$\mathtt{T2} \rightleftarrows \mathtt{T1}$

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.

```
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)`

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)

two-phase locking (2PL)

1. each shared variable has a 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

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

two-phase locking (2PL)

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

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

two-phase locking (2PL)

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

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

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

two-phase locking (2PL)

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

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

```
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)

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

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

```
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)

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

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

```
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 **cannot** do this; it breaks the third rule of 2PL

two-phase locking (2PL)

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

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

```
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)

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

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

```
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)

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

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

```
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)

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

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

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \dots \rightarrow T_k$$

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

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_k$

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

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

$$\begin{tikzcd} T_1 \arrow[r, "x_1"] & T_2 \arrow[r, "x_2"] & T_3 \arrow[r, "x_3"] & \dots \arrow[r, "x_{k-1}"] & T_k \arrow[llll, bend left, "x_k"] \end{tikzcd}$$

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

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

$$\begin{tikzcd} T_1 \arrow[r, "x_1"] & T_2 \arrow[r, "x_2"] & T_3 \arrow[r, "x_3"] & \dots \arrow[r, "x_{k-1}"] & T_k \arrow[llll, bend left, "x_k"] \end{tikzcd}$$

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

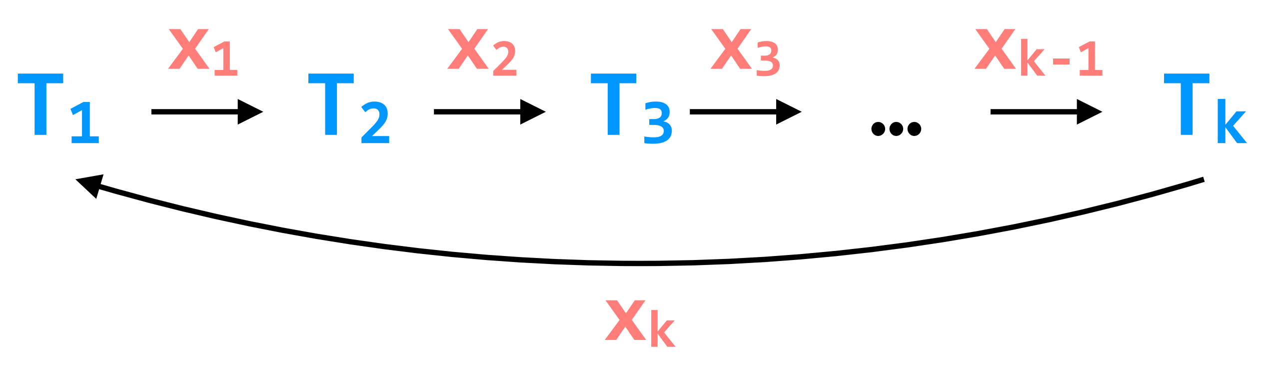
two-phase locking (2PL)

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

```
T₁ acquires x₁.lock
T₂ acquires x₁.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

$$\begin{tikzcd} T_1 \arrow[r, "x_1"] & T_2 \arrow[r, "x_2"] & T_3 \arrow[r, "x_3"] & \dots \arrow[r, "x_{k-1}"] & T_k \arrow[llll, bend left, "x_k"] \end{tikzcd}$$

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

T_1 **acquires** x_1**.lock**
T_2 **acquires** x_1**.lock**

T_2 **acquires** x_2**.lock**
T_3 **acquires** x_2**.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

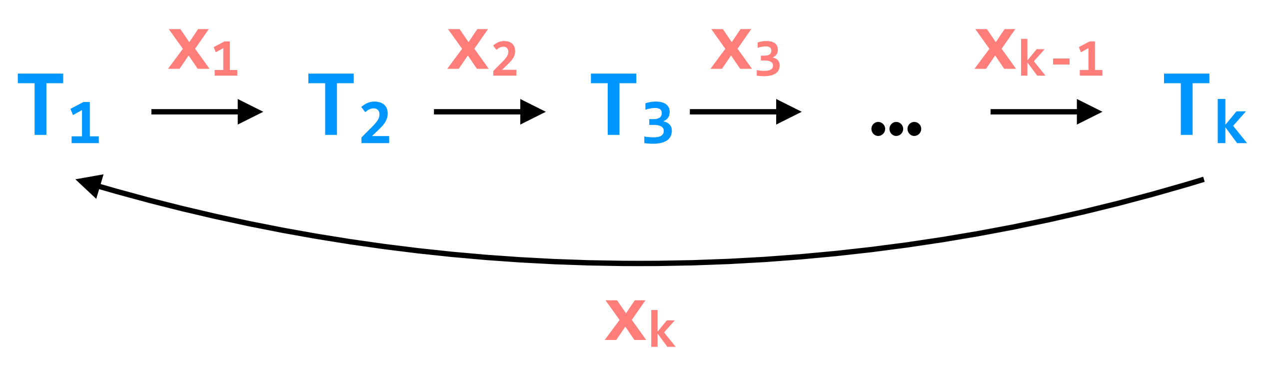
two-phase locking (2PL)

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



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

T_1 acquires x_1.lock
T_2 acquires x_1.lock

T_2 acquires x_2.lock
T_3 acquires x_2.lock

…

T_k acquires x_k.lock
T_1 acquires x_k.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

$$\begin{tikzcd} T_1 \arrow[r, "x_1"] & T_2 \arrow[r, "x_2"] & T_3 \arrow[r, "x_3"] & \dots \arrow[r, "x_{k-1}"] & T_k \arrow[llll, bend left, "x_k"] \end{tikzcd}$$

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

T_1 **acquires** x_1**.lock**
T_2 **acquires** x_1**.lock**

T_2 **acquires** x_2**.lock**
T_3 **acquires** x_2**.lock**

…

T_k **acquires** x_k**.lock**
T_1 **acquires** x_k**.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

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

$$\begin{tikzcd} T_1 \arrow[r, "x_1"] & T_2 \arrow[r, "x_2"] & T_3 \arrow[r, "x_3"] & \dots \arrow[r, "x_{k-1}"] & T_k \arrow[llll, bend left, "x_k"] \end{tikzcd}$$

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

T_1 **acquires** x_1**.lock**
T_1 **releases** x_1**.lock**
T_2 **acquires** x_1**.lock**

T_2 **acquires** x_2**.lock**
T_3 **acquires** x_2**.lock**

…

T_k **acquires** x_k**.lock**
T_1 **acquires** x_k**.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

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

$$\begin{tikzcd} T_1 \arrow[r, "x_1"] & T_2 \arrow[r, "x_2"] & T_3 \arrow[r, "x_3"] & \dots \arrow[r, "x_{k-1}"] & T_k \arrow[llll, bend left, "x_k"] \end{tikzcd}$$

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

T_1 **acquires** x_1**.lock**
T_1 **releases** x_1**.lock**
T_2 **acquires** x_1**.lock**

T_2 **acquires** x_2**.lock**
T_3 **acquires** x_2**.lock**

...

T_k **acquires** x_k**.lock**
T_1 **acquires** x_k**.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

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

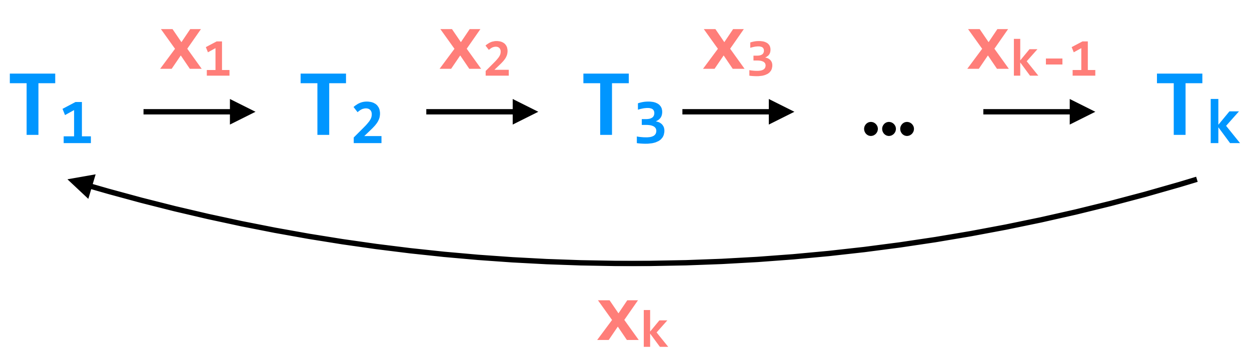
two-phase locking (2PL)

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

```
T1 acquires x1.lock
T1 releases x1.lock
T2 acquires x1.lock

T2 acquires x2.lock
T3 acquires x2.lock

...

Tk acquires xk.lock
T1 acquires xk.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

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

T_1 released a lock (x_1`.lock`) and then acquired a lock (x_k`.lock`) afterwards

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

```
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(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)
```

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

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; 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: 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; a better solution is to **take advantage of atomicity and abort one of the transactions**

we can detect deadlock in a number of ways — for example, build a graph of which transactions are waiting for each other and use that, or abort a transaction as soon as it blocks when attempting to acquire a lock. these techniques have performance trade-offs; techniques that perform better typically also sometimes abort transactions that didn't need to be aborted

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

```
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: 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

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**

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

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

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.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)
```

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

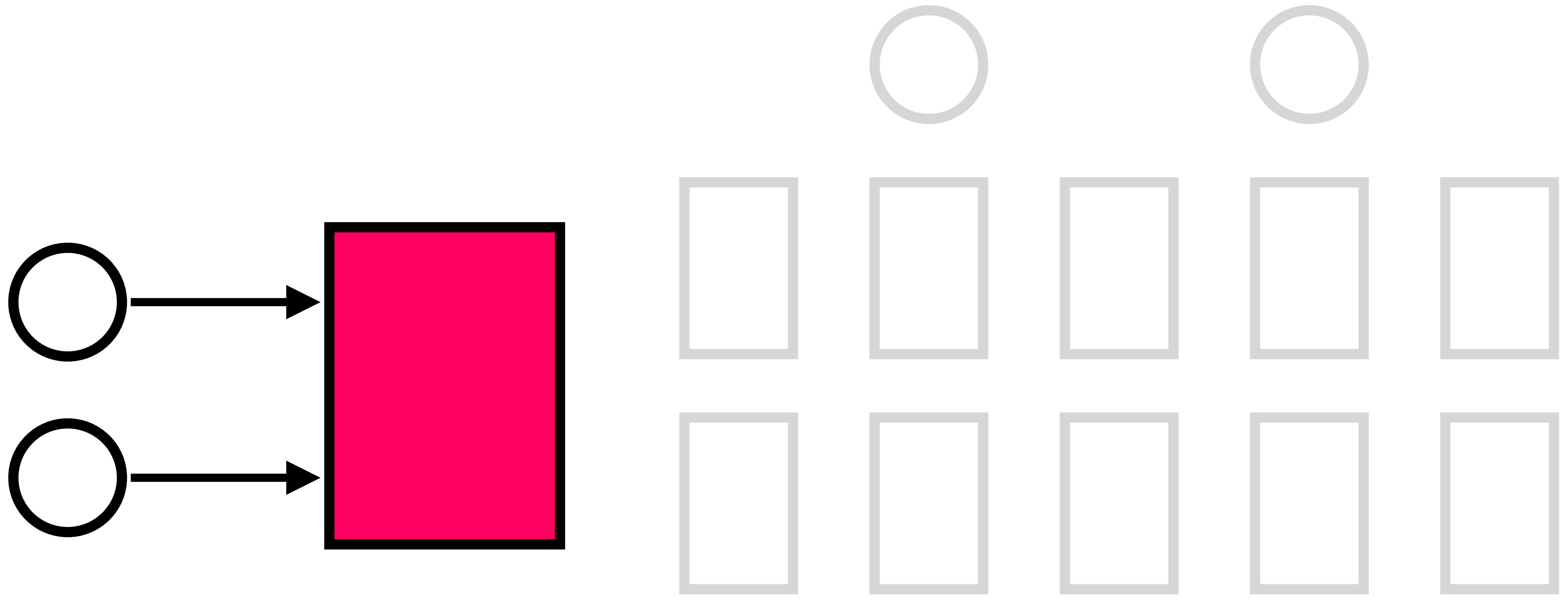
problem: performance

```
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

* shadow copies *are* used in some systems

isolation: provided by **two-phase locking**

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