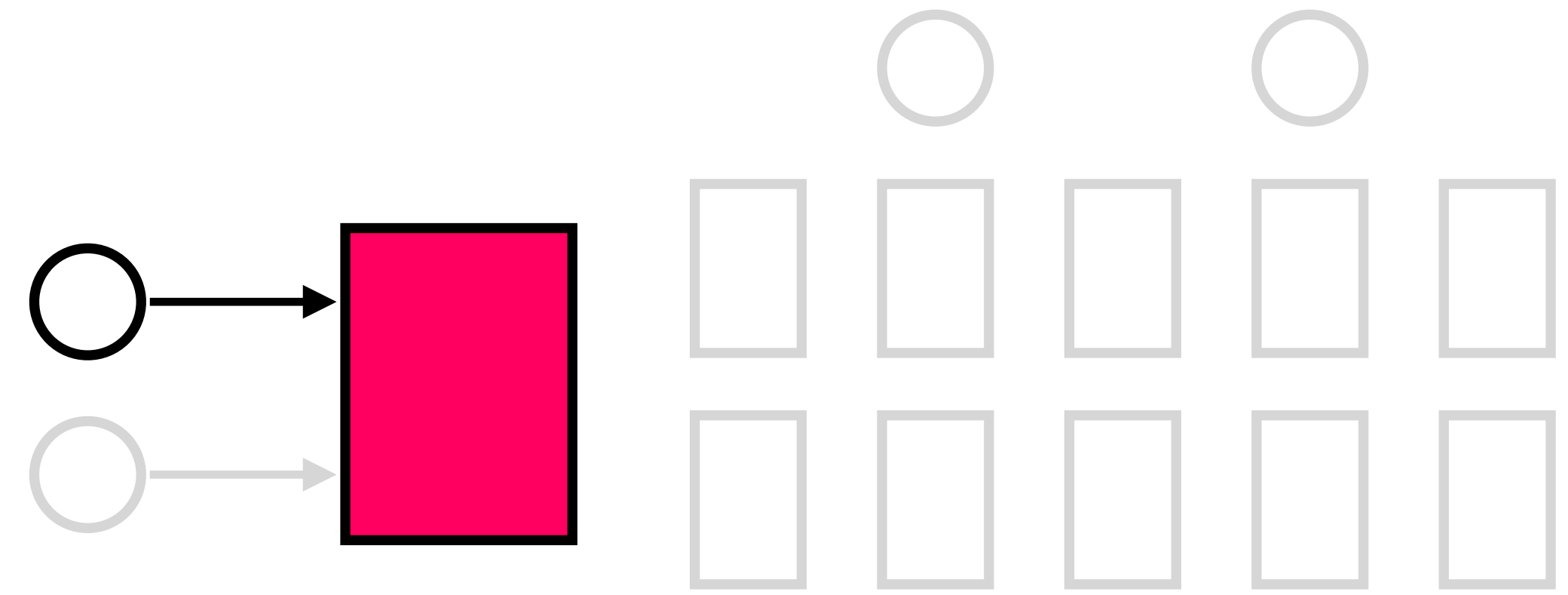


6.033 Spring 2021

Lecture #17: 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

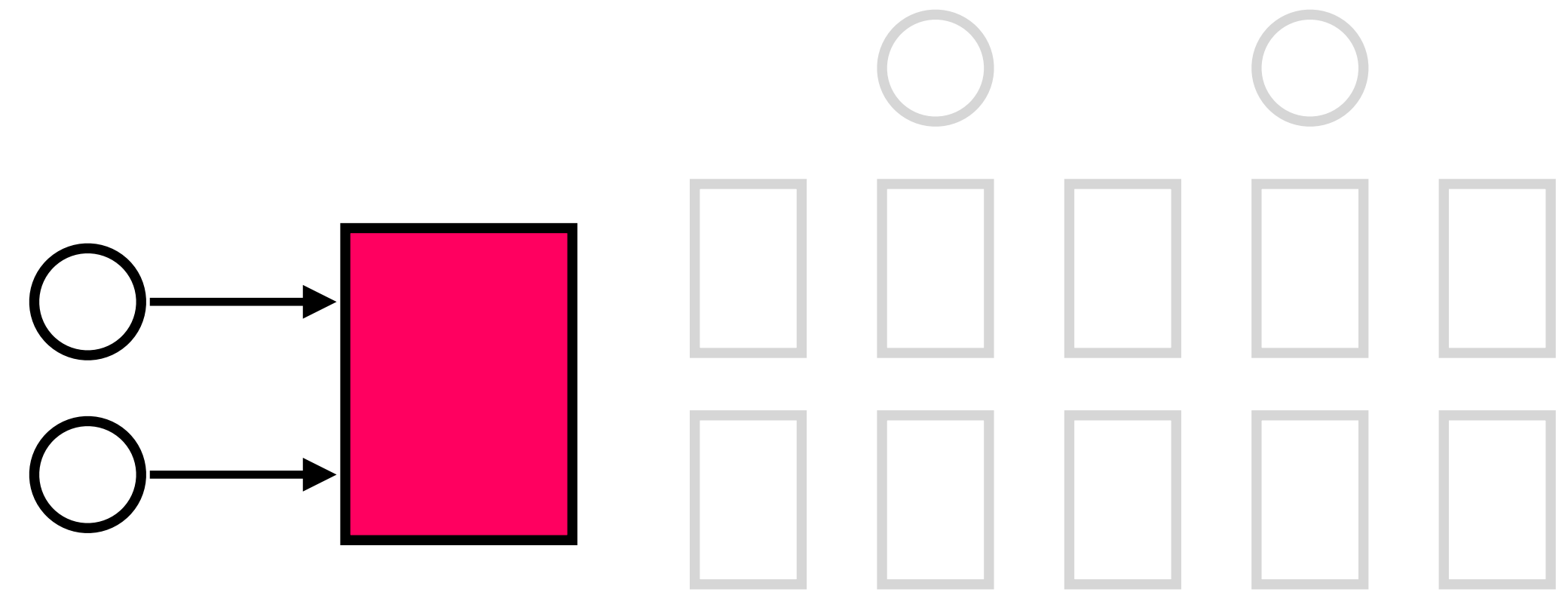
our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

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

* 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

↖ what does this even mean?

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** 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)

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
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 a 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

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

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

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

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

this schedule is conflict serializable

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

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

T2 \rightarrow **T1**

this schedule is conflict serializable

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

T2 \leftrightarrow **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)

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)

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

often we release locks after commit, which 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

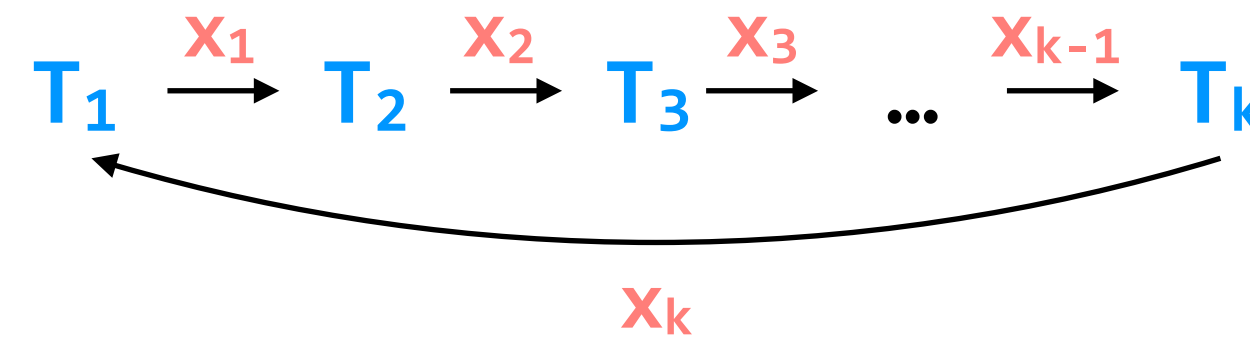
two-phase locking (2PL)

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)

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



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

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**

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

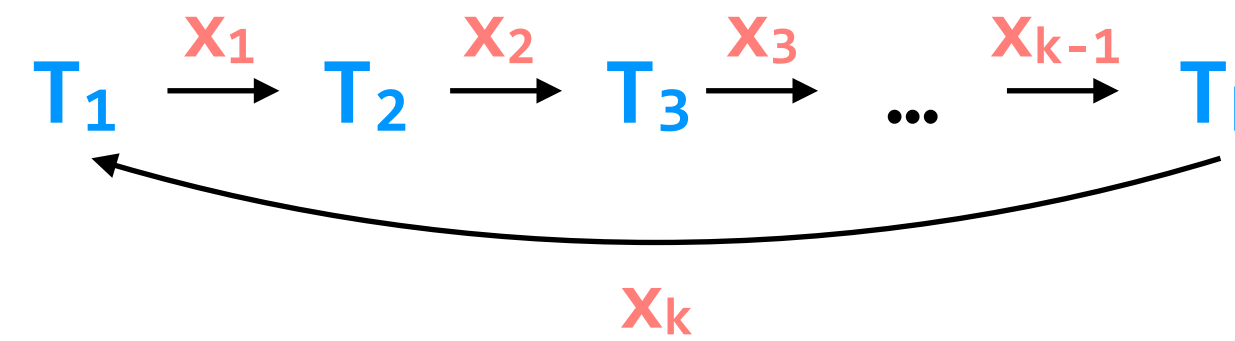
two-phase locking (2PL)

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

...

T_k acquires x_k.lock
T₁ acquires x_k.lock

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

in order for the schedule to progress, **T₁** must have released its lock on **x₁** before **T₂** acquired it

contradiction: this is not a valid 2PL schedule

two-phase locking (2PL)

problem: 2PL can result in deadlock

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	T2
begin	begin
acquire(x.lock)	acquire(y.lock)
T1.1 read(x)	T2.1 write(y, 30)
acquire(y.lock)	acquire(x.lock)
T1.2 tmp = read(y)	T2.2 write(x, 20)
T1.3 write(y, tmp+10)	commit
commit	release(x.lock)
release(x.lock)	release(y.lock)
release(y.lock)	

for example, suppose T2 wrote to y before x

one solution to this problem is a global ordering on locks; but we hate that! a better solution is to take advantage of atomicity and abort one of the transactions

two-phase locking (2PL)

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

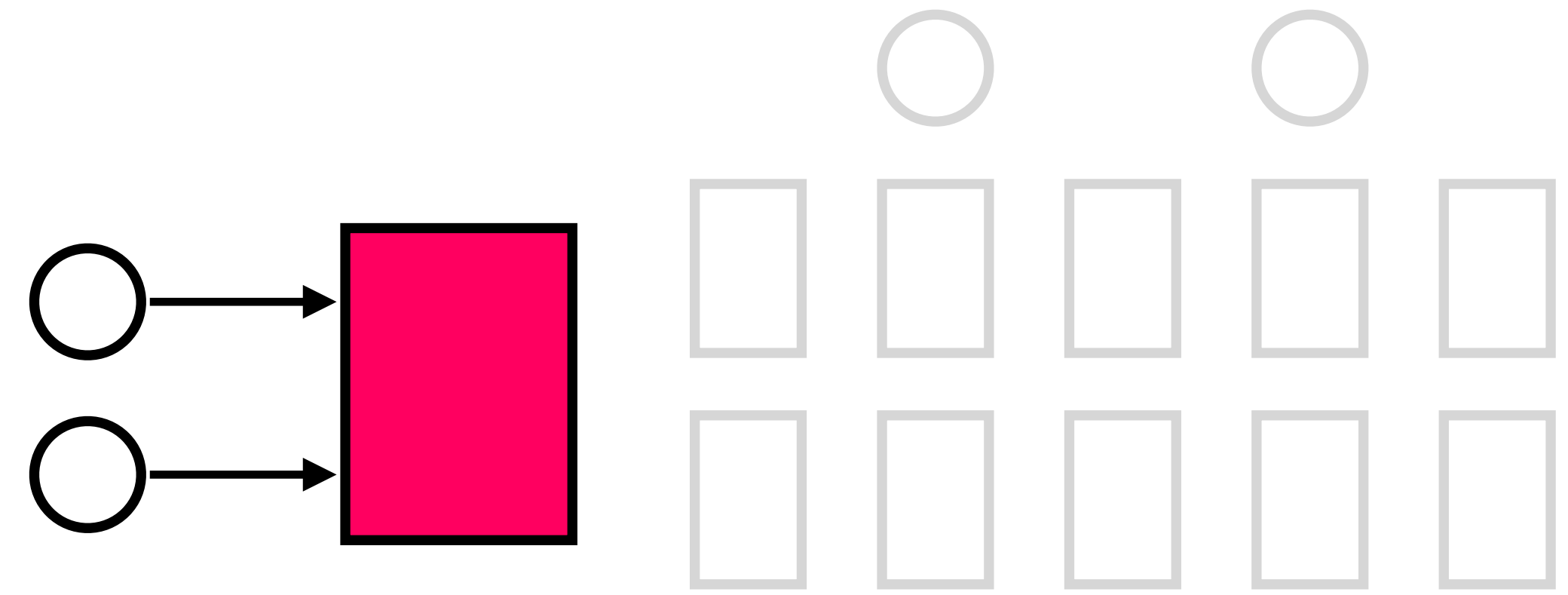
problem: performance

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