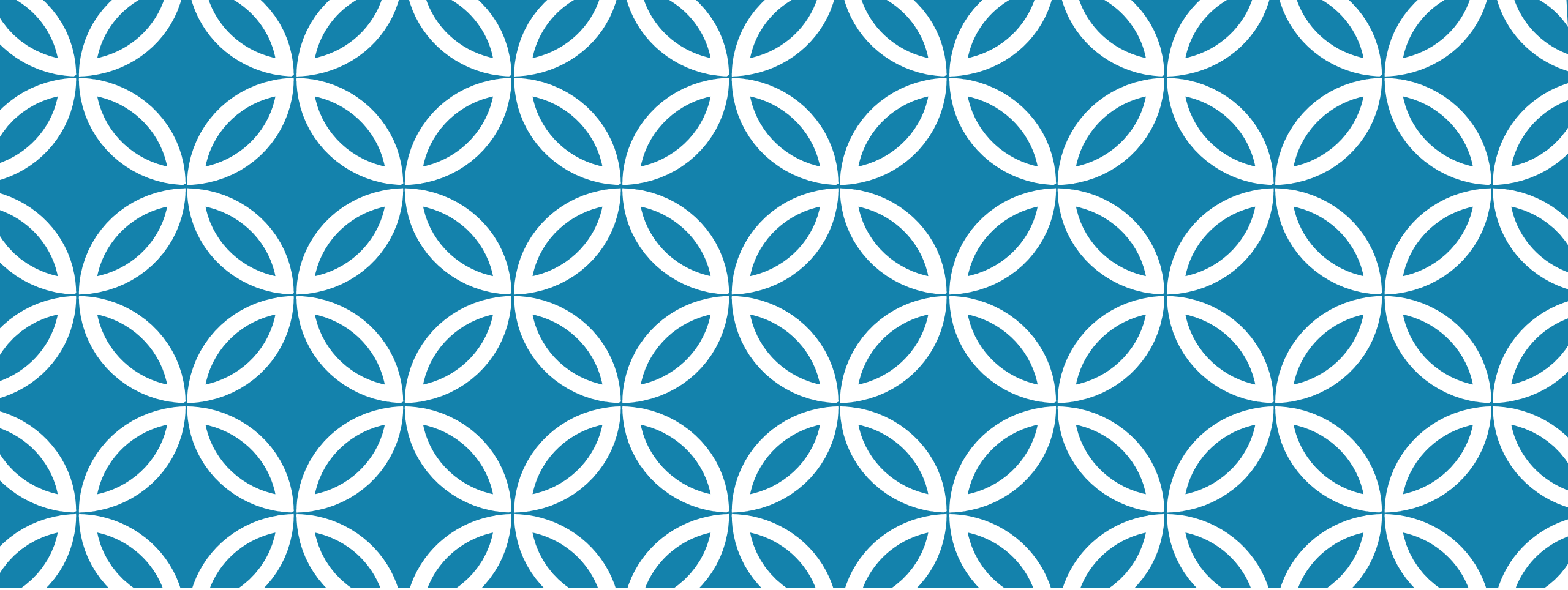


CS4414: RECITATION 13 – TRANSACTIONS

Ricky Takkar

Friday, April 28, 2023

PRELIM 2 (== FINAL) NEXT WEEK!

- Day: May 2 (Tuesday). Time: 7:30pm -> 10:00pm. Location: Room KG70, Klarman Hall
- Exam will be just like prelim1 in terms of format and length
- Coverage will be up through and including the lecture on April 26
- Just like for prelim1 it will be a closed book exam, but we will allow you to bring a page of your own notes (on paper, standard sized, both sides allowed, typed or handwritten) if you wish

TRANSACTIONS 101

- **Transaction model**: a way to describe correct, consistent behavior when distributed programs concurrently access storage that could be spread over many machines.
- **Two-Phase commit**: a central building block for a solution. Ensures that if any process commits, all do; otherwise it aborts.
- **Two-phase locking** (similar name, totally different meaning!): A way to do read and write locking that, when combined with two-phase commit, ensures transactional serializability

https://en.wikipedia.org/wiki/Memcached

ROLE OF BEGIN AND COMMIT/ABORT? (REVIEW L24)

Begin is a kind of a “curly brace”. But in fact it denotes the place where the transactional system initializes itself.

Commit is the way a successful transaction tells the runtime environment to save (make permanent) all its changes.

Abort tells the system to back the changes out.

DATA AND PROCESSES (REVIEW L24)

We model data as a set of variables, usually with alphabetical names such as X, Y, Z…

A transaction models an executing program that has begin/commit/abort blocks, inside of which it issues reads and writes to the variables.

SYNCHRONIZATION (REVIEW L24)

We expect to have lots of concurrent processes running, so we need a way to avoid concurrency issues.

For this a transactional model introduces read locks and write locks. If you hold a read lock on X, you can **only do reads.** With a write lock, you can do **both reads and writes.**

BASIC PHILOSOPHY (REVIEW L24)

Our concurrent system should behave just like it ran one transaction at a time, to completion, then started the other.

But the order in which they run isn't predictable. Any permuted order is considered to be a correct run of the system.

This property is called *serializability*.

EXECUTION TRACE: T_1 RUNS FIRST, THEN T_2

(REVIEW L24)

Transaction 1:

```
Begin;
    ReadLock X;
    ReadLock Y;
    WriteLock Z;
    Z = X+Y;
Commit;
```

Transaction 2:

```
Begin;
    ReadLock Z;
    WriteLock X;
    WriteLock Y;
    X = Y-Z;
    Y = X+Z;
Commit;
```

In this trace, time goes from left to right

SECOND EXAMPLE: T_2 RUNS FIRST, THEN T_1

(REVIEW L24)

Transaction 1:

```
Begin;
    ReadLock X;
    ReadLock Y;
    WriteLock Z;
    Z = X+Y;
Commit;
```

Transaction 2:

```
Begin;
    ReadLock Z;
    WriteLock X;
    WriteLock Y;
    X = Y-Z;
    Y = X+Z;
Commit;
```

In this trace, time goes from left to right

THIRD TRACE: INTERLEAVED. IS THIS A SERIALIZABLE EVENT ORDERING? (REVIEW L24)

Transaction 1:

```
Begin;
    ReadLock X;
    ReadLock Y;
    WriteLock Z;
    Z = X+Y;
Commit;
```

Transaction 2:

```
Begin;
    ReadLock Z;
    WriteLock X;
    WriteLock Y;
    X = Y-Z;
    Y = X+Z;
Commit;
```

WAS THE THIRD TRACE SERIALIZABLE? (REVIEW L24)

Suppose initially X=1, Y=2, Z=9

First trace:

T_1 leaves X=1, Y=2, Z=3

… then T_2 leaves X=-1, Y=2, Z=3

DO THESE TRACES GIVE CORRECT RESULTS? (REVIEW L24)

Suppose initially X=1, Y=2, Z=9

First trace:

T_1 leaves X=1, Y=2, Z=3

… then T_2 leaves **X=-1, Y=2, Z=3**

Now consider trace 2 for X=1, Y=2, Z=9

Here, T_2 ran first, then T_1

T_2 leaves X=-7, Y=2, Z=9

… then T_1 leaves **X=-7, Y=2, Z=-5**

Bold: these outcomes reflect the two possible orderings

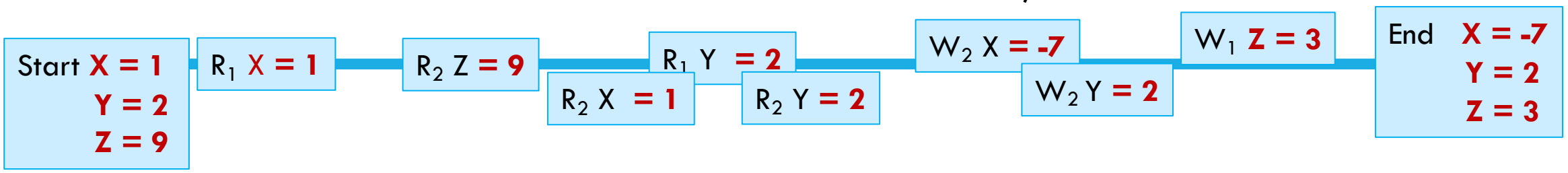
HAND-COMPUTING THE INTERLEAVED OUTCOME (REVIEW L24)

Transaction 1:

```
Begin;
    ReadLock X;
    ReadLock Y;
    WriteLock Z;
    Z = X+Y;
Commit;
```

Transaction 2:

```
Begin;
    ReadLock Z;
    WriteLock X;
    WriteLock Y;
    X = Y-Z;
    Y = X+Z;
Commit;
```



DO THESE TRACES GIVE CORRECT RESULTS? (REVIEW L24)

We started with X=1, Y=2, Z=9

The T_1T_2 serialization order results in:

X=-1, Y=2, Z=3

The T_2T_1 serialization order results in:

X=-7, Y=2, Z=-5

… But the interleaved execution results in:

X=-7, Y=2, Z=3

This can't happen with the ordering T_1 T_2 or T_2 T_1

A FAMILIAR SITUATION! JUST LIKE CRITICAL SECTIONS WITH INTERFERENCE! (REVIEW L24)

… It turns out that serialized orderings make sense, but non-serialized execution orderings are almost always nonsense.

We need to allow concurrency (for speedup) but prevent disordered/scrambled outcomes.

Idea: we need a way to enforce serializability

ACID MODEL, SERIALIZABILITY (REVIEW L24)

Jim Gray and others proposed a simple set of rules to describe how transactions should behave: ACID

- **A**tomic: All or nothing.
- **C**onsistent: A correct transaction takes the data from one consistent state to another consistent state.
- **I**solation: If two transactions run at the same time, they should see one-another's pending (uncommitted) updates.
- **D**urability: Once committed, updates won't get lost.

TWO-PHASE COMMIT (REVIEW L24)

A “distributed protocol” aimed at solving a practical issue seen with transactions when data is spread over multiple servers.

Suppose that X and Y and Z are each held by different servers. When a transaction runs, it creates pending updates, X’, Y’, Z’. Commit makes these permanent… Abort would roll them back.

But how do we ensure “all or nothing” commit (or abort)?

TWO-PHASE COMMIT (REVIEW L24)

1. T says to X, Y and Z: are you able to commit?
2. X and Y and Z **must first log X' and Y' and Z' on disk.** This is to ensure that even with a crash, they are still prepared to commit.
3. Then each replies: "I'm prepared to commit!"
4. T can commit if all three are prepared… but should abort if any doesn't respond or replies that it "must abort".
5. T also logs its decision, so if Y is down when T commits, later Y can find out what it should do. *We call this an* ***outcomes log.***
6. *Step 5 assumes the log is highly available, but there are ways to ensure this.*

PROBLEM SOLVED! (REVIEW L24)

With two-phase commit, either all of the servers (eventually) commit and install the update, or all of them abort.

A crashed server will reboot with the update still pending, but won't have lost it. So by checking the outcomes log, it learns that the transaction committed, and then it finalizes the outcome before resuming participation in the system. "Automatic repair"!

See L24: slide 27 onward for more on how to deal with **locking…**

PRELIM 2 PREP

Shared Google doc

- Doc
- 10-15 mins for coming up with questions
- 10-15 mins for coming up with answers

Practice Final

- 2021 Exam
- 2021 Solution