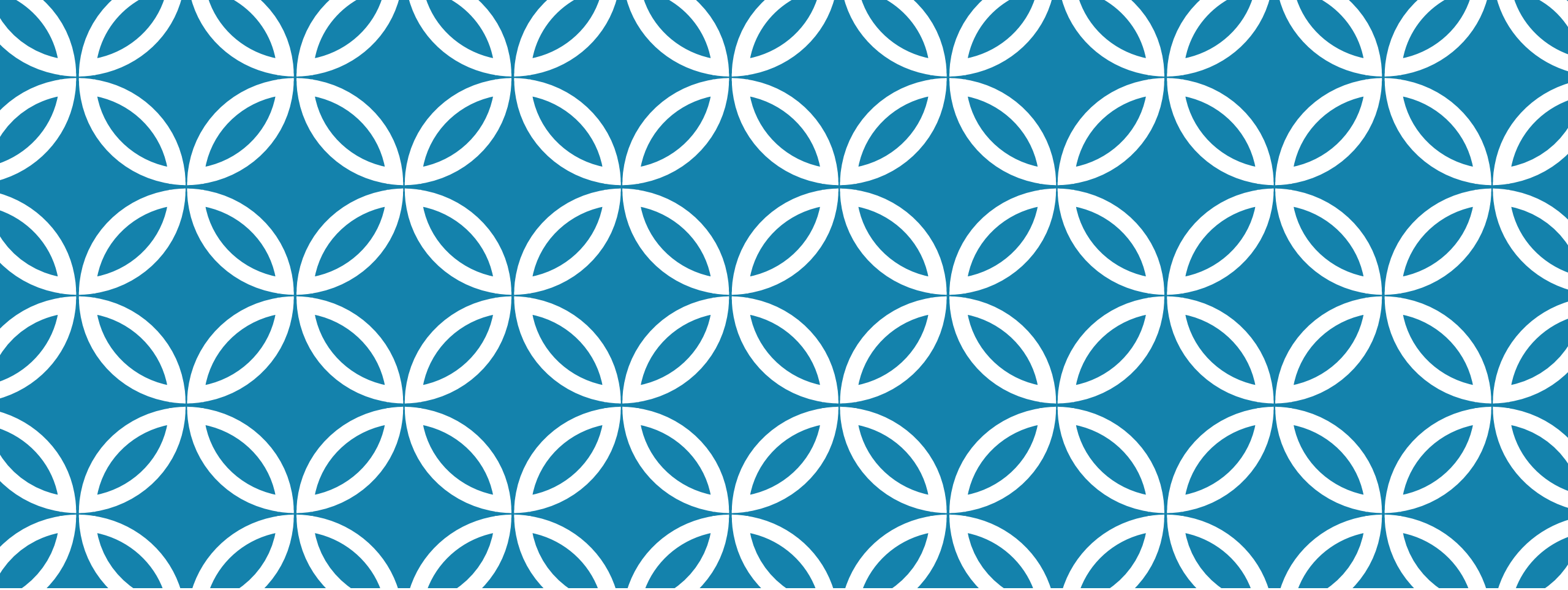


FINISH SOME LEFTOVER C++ TOPICS THEN: DEADLOCKS, LIVELOCKS

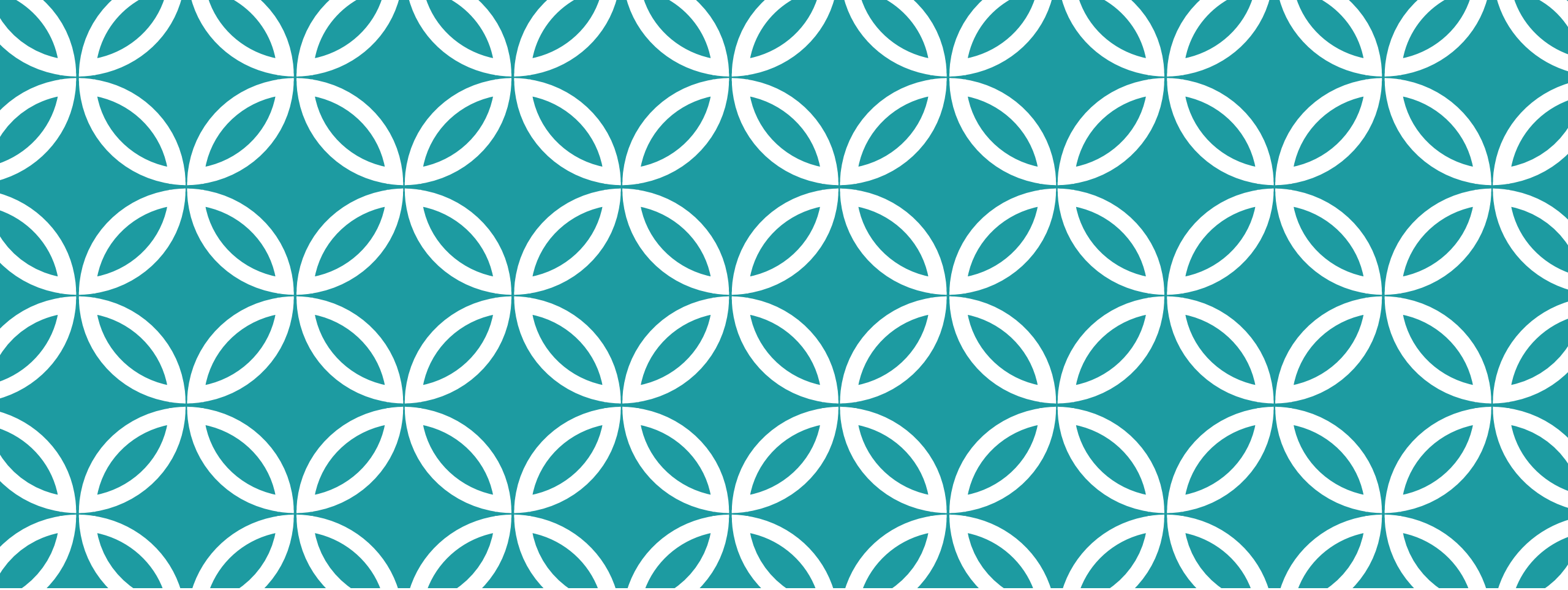
Professor Ken Birman

CS4414 Lecture 16

BEFORE WE DIVE IN…

First, a “left over” mini-topic:

A quick glimpse of booling: A way to do database and file system access from C++ that leads to really nice looking code



LINQ FOR C++ (BOOLINQ)...

LINQ AND BOOLINQ

LINQ: A family of higher-level templated librarys that support database access or scaning collections of files.

Uses a notation popular in ML systems (seen in Pandas/NumPy dialect of Python, Tensor Flow, Spark/Databricks, Julia…).

https://xscode.com/k06a/booling

KEY IDEA

Connect to database, file system or “key-value” storage.

Obtain a collection that’s supports iterators.

Now you can just write expressions that look like database expressions anywhere in your C++ code, and they can mix C++ and database operators very easily.

ITERATORS AND PAIRS

LINQ centers on:

- (key, value) pairs. A key could just be a name, a file path, or any unique id.

  Example: for a database the key is a row-id, value is the row

- A collection could be std::list, std::map, etc.
- *Iterators*: C++ object used in for loops to scan a collection, or a range within a collection.

LINQ EXAMPLES

Things to notice:

- Code is very "succinct"
- Lots of use of lambdas
- Very powerful
- Mixes with normal C++ (in fact, is a C++ library)

Sum the even numbers from an array of integers:

```
int src[] = {1, 2, 3, 4, 5, 6, 7, 8};
auto dst = from(src)
        .where( [](int a) { return a % 2 == 1; })  // 1, 3, 5, 7
        .select([](int a) { return a * 2; })       // 2, 6, 10, 14
        .where( [](int a) { return a > 2 && a < 12; }) // 6, 10
        .toStdVector();     // dst will be a std::vector with 6, 10
```

Order descending all the distinct numbers from an array of integers, transform them into strings and print the result.

```
int numbers[] = {3, 1, 4, 1, 5, 9, 2, 6};
auto result = from(numbers)
          . distinct()
          . orderby_descending([](int i) {return i;})
          . select([](int i){std::stringstream s; s<<i; return s.str();})
          . toStdVector();
for(auto i : result)
    std::cout << i << std::endl;
```

Visit booling on GitHub to download, learn more

EXAMPLE WITH STRUCTS

In a list of friends, find the subset who are under age 18:

```
struct Friends { std::string name; int age; };

Friends src[] = {
   {“Kevin”, 14}, {“Anton”, 18}, {“Agata”, 17}, “Saman”, 20}, {“Alice”, 15},
};

auto dst = from(src).where([](const Friends & who) { return who.age < 18; })
                    .orderBy([](const Friends & who) { return who.age; })
                    .select( [](const Friends & who) { return who.name; })
                    .toStdVector();

// dst type: std::vector<:string>…  items: “Kevin”, “Agata”, “Alice”
```

EXAMPLE WITH STRINGS

In a list of text messages, count the number of messages to Dennis by sender:

```
struct Message {  std::string PhoneA; std::string PhoneB;  std::string Text;  };

Message messages[] = {
    {“Anton”,  “Troll”, “Hello, friend!”},
    {“Denis”,  “Mark”,  “Join us to watch the game?"},
    {“Anton”,  “Sarah”, “OMG! ”},
    {“Denis”,  “Jimmy", “How r u?”},
    {“Denis”,  “Mark",  “The night is young!”},
};

int DenisUniqueContactCount =
    from(messages)
        .where([](const Message & msg) { return msg.PhoneA == “Denis”; })
        .distinct([](const Message & msg) { return msg.PhoneB; })
        .count();
```

SOME LINQ OPERATORS

Filters and reorders:

- `where(predicate)`, `where_i(predicate)`
- `take(count)`, `takeWhile(predicate)`, `takeWhile_i(predicate)`
- `skip(count)`, `skipWhile(predicate)`, `skipWhile_i(predicate)`
- `orderBy()`, `orderBy(transform)`
- `distinct()`, `distinct(transform)`
- `append(items)`, `prepend(items)`
- `concat(linq)`
- `reverse()`
- `cast()`

Transformers:

- `select(transform)`, `select_i(transform)`
- `groupBy(transform)`
- `selectMany(transfom)`

`Bits and Bytes:`

- `bytes(ByteDirection?)`
- `unbytes(ByteDirection?)`
- `bits(BitsDirection?, BytesDirection?)`
- `unbits(BitsDirection?, BytesDirection?)`

Aggregators:

- `all()`, `all(predicate)`
- `any()`, `any(lambda)`
- `sum()`, `sum()`, `sum(lambda)`
- `avg()`, `avg()`, `avg(lambda)`
- `min()`, `min(lambda)`
- `max()`, `max(lambda)`
- `count()`, `count(value)`, `count(predicate)`
- `contains(value)`
- `elementAt(index)`
- `first()`, `first(filter)`, `firstOrDefault()`, `firstOrDefault(filter)`
- `last()`, `last(filter)`, `lastOrDefault()`, `lastOrDefault(filter)`
- `toStdSet()`, `toStdList()`, `toStdDeque()`, `toStdVector()`

Coming soon:

- gz(), ungz(), leftJoin, rightJoin, crossJoin, fullJoin

HOW TO "CONNECT" TO A DATABASE LIKE MYSQL OR ORACLE

LINQ requires a "connector" but you won't have to build it: databases, file systems and key-value stores provide these.

You specify the name of the database and the connector returns a collection object that supports iterators. So simply by constructing a connection you can access the data in LINQ.

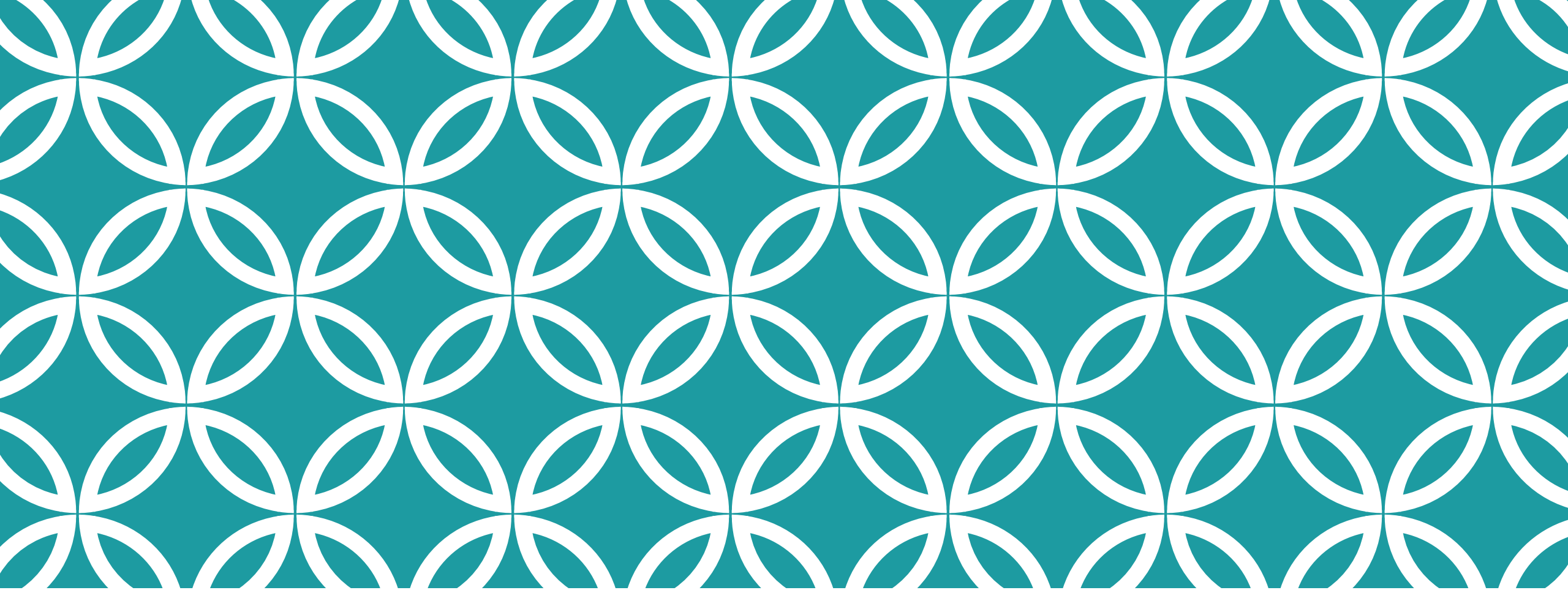
Example: In MySQL, you could use the X-DevAPI.

HOW DOES THIS TIE INTO C++ FOR ML?

Many machine learning systems are trained on data in vectors, arrays or higher-dimensional tensors.

A database query returns a table as a result. Think of the table as a collection of (row-id, row-contents) pairs. Easy to perform in LINQ

Finally, we pass the data to ML algorithms expressed as matrix multiplications, eigenvalue computations, etc. We end up with ML code in a high-level form that executes extremely efficiently.



NEXT: OUR MAIN TOPIC…

IDEA MAP FOR THE REST OF OUR LECTURE

Reminder: Thread Concept

Lightweight vs. Heavyweight

Thread "context"

C++ mutex objects. Atomic data types.

The monitor pattern in C++

Problems monitors solve (and problems they don't solve)

Deadlocks and Livelocks

Today we focus on deadlocks and livelocks.

DEADLOCK: UNDERSTANDING

Deadlock arises in situations where we have multiple threads that share some form of protected object or objects.

For simplicity, A and B share X and Y.

Now suppose that A is holding a lock on X, and B has a lock on Y. A tries to lock Y, and B tries to lock X. Both wait, forever!

MORE EXAMPLES

We only have one object, X.

A locks X, but due to a caught exception, exits the lock scope. Because A didn't use scoped_lock, the lock isn't released.

Now B tries to lock X and waits. Because A no longer realizes it holds the lock, this will persist forever.

ACQUIRING A MUTEX "TWICE"

Suppose that A is in a recursive algorithm, and the same thread attempts to lock mutex X more than once. The recursion would also unlock it the same number of times.

This is possible with a C++ "recursive_mutex" object.

But the standard C++ mutex is not recursive.

WHAT IF YOU TRY TO RECURSIVELY LOCK A NON-RECURSIVE MUTEX?

The resulting behavior is not defined.

On some platforms, this will deadlock silently. **A waits for A!**

On others, you get an exception, "Deadlock would result."

MORE EXAMPLES

A and B lock X and Y. The developer noticed the deadlock pattern but did not understand the issue.

C++ lock primitives have optional "timeout" arguments. So the developer decided to add a "random backoff" feature:

- When locking an object, wait ***t*** milliseconds.
- Initially, ***t=0*** but after a timeout, change to a random value [0..999]
- Then retry

WHAT DOES THIS GIVE US?

Now A locks X (and holds the lock), and B locks Y

A tries to lock Y, times out, retries… forever

B tries to lock X, times out, retries… forever

They aren't "waiting" yet they actually *are* waiting!

DEADLOCK AND LIVELOCK DEFINITIONS

We say that a system is in a deadlocked state if one or more threads will wait indefinitely (for a lock that should have been released).

Non-example: A is waiting for input from the console. But Alice doesn't type anything.

Non-example: A lock is used to signal "a cupcake is ready", but we have run out of sugar and none can be baked.

NECESSARY AND SUFFICIENT CONDITIONS FOR DEADLOCK

1. **Mutual exclusion:** The system has resources protected by locks
2. **Non-shareable resources:** while A holds the lock, B waits.
3. **No preemption:** there is no way for B to "seize the lock" from A.
4. **Cyclic waiting:** A waits for B, B waits for A (a "circular" pattern)

With recursion using non-recursive locks, A could deadlock "by itself"

CONDITIONS FOR LIVELOCK

A livelock is really the same as a deadlock, except that the threads or processes have some way to "spin".

As a result, instead of pausing, one or more may be spin-waiting.

We can define "inability to enter the critical section" as a wait, in which case the four necessary and sufficient conditions apply.

C++ AND LINUX ARE FULL OF RISKS!

If you think about it, you can find hundreds of ways that Linux could potentially be at risk of deadlocks!

If you code with threads in C++ you run that risk too!

The developers of Linux designed the system to be free of deadlock. You can do so in your applications too. But it takes conscious though and a careful design.

HOW TO AVOID DEADLOCKS!

Acquire locks in a fixed order that every thread respects. This rule implies that condition 4 (cyclic waiting) cannot arise.

Example: Recall A and B with X and Y.

- We had A holding a lock on X and requesting a lock on Y: if our rule says lock X before Y, this is legal and A must wait.
- Meanwhile B held a lock on Y. Given our rule, B is not allowed to request a lock on X at this point.

… THIS RULE CAN BE IMPRACTICAL

There are many applications that learn what they must lock one item at a time, in some order they cannot predict.

So in such a situation, B didn't know it would need a lock on X at the time it locked Y.

… now it is too late!

… ON THE OTHER HAND, THE RULE IS USEFUL

When you actually *can* impose an order and respect the rule, it is a very simple and convenient way to avoid deadlock.

Ordered locking is very common inside the Linux kernel. It has a cost (an application may need to sort a list of items, for example, before locking all of them), but when feasible, it works.

TIMER BASED SOLUTIONS

Sometimes it is too complicated to implement orderd locking.

So we just employ a timeout.

If B is running and tries to get a lock, but a timeout occurs, B aborts (releasing all its locks) and restarts.

BACKING OUT AND RETRYING

Backout can be costly

For this purpose, B would employ "try_lock".

This is a feature that acquires a lock if possible within some amount of time, but then gives up.

If B gets lucky, it is able to lock Y, then X, and no deadlock arises. But if the lock on Y fails, B must unlock X.

CONCEPT: ABORT AND RETRY

We say that a computation has "aborted" if it has a way to undo some of the work it has done.

For example, B could be executing, lock Y, then attempt to lock X. The try_lock fails, so B releases the lock on X and throws away the temporary data it created – it "rolls back". Then it can retry, but get a lock on X first. Hopefully this will succeed.

DOES THIS WORK?

Many database systems use abort/retry this way.

If deadlocks are very rare, the odds are that on retry, B will be successful.

But if deadlocks become common, we end up with a livelock.

PREEMPTIVE SOLUTION ("WOUND-WAIT")

This method requires some way for the system to detect a deadlock if one arises, and a way for threads to abort.

When A and B start executing, each notes its start time.

Rule: in a deadlock, **the older thread wins.** So if A was first, A gets to lock Y and B aborts. If B was older, A aborts.

DETECTING DEADLOCKS

Clearly, we gain many options if a system has a way to detect deadlocks. Does C++ support this?

… you might think so, given the "deadlock would arise" exception for recursive locking. But in fact this is done just by tracking the thread-id for the thread holding a mutex.

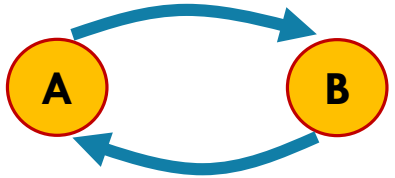
HOW TO BUILD A DEADLOCK DETECTOR

We wrap every locking operation with a method that builds a graph of which thread is waiting for which other thread.

For example, if A tries to lock Y, but B is holding that lock, we add a node for A, a node for B, and an A $\rightarrow$ edge.

If a thread is waiting for long enough, run "cycle detection".

CYCLE DETECTION ALGORITHMS



Run the depth-first search algorithm.

Back-edges imply a cycle; success with no back-edges implies that the graph is cycle-free, hence there is no deadlock.

Complexity: V+E, where V is the number of threads (nodes) and E is the number of wait-edges.

PRIORITY INVERSIONS

In some systems, threads are given different priorities to run.

- Urgent: The thread should be scheduled as soon as possible.
- Normal: The usual scheduling policy is fine.
- Low: Schedule only when there is nothing else that needs to run.

A priority inversion occurs if a higher priority thread is waiting for a lower priority thread.

Deadlock can now arise if there is a steady workload of high priority tasks, so that the lower priority thread doesn't get a chance to run.

HOW TO DETECT THIS SORT OF PROBLEM

If we create a deadlock detector, we can extend it do handle priority-inversion detection!

For each mutex, track the priority of any thread that accesses it.

If we ever see a mutex that is accessed by a high and a low priority thread, a risk of priority inversion arises!

WHAT TO DO ABOUT IT?

One option is to temporarily change the priority of the lower priority thread.

Suppose that A holds a mutex on X.

B, higher priority than A, wants a lock on X. We can "bump" A to higher priority temporarily, then restore A to lower priority when it releases the lock on X.

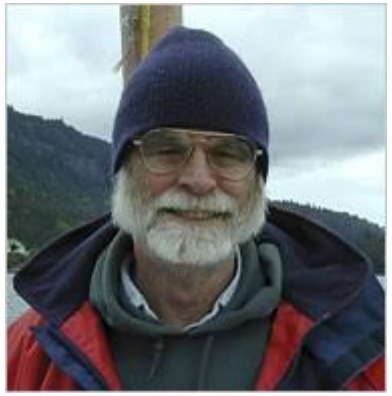
NONE OF THESE IS CHEAP…

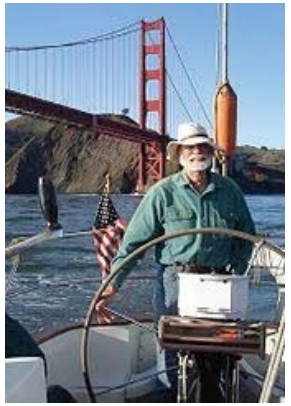
Recall our discussion of C++ versus Java and Python.

These methods of watching for cycles or priority inversions, possibly forcing threads to abort, rollback and retry, etc, are all examples of runtime mechanisms that can be very costly!

If you have no choice, then you use them. But don't be naïve about how expensive they can become!

JIM GRAY'S STUDY





In the 1990's, databases were used for storing all forms of data

By the early 2000's, they became extremely big and heavily loaded. People began to move them to NUMA machines and to use lots of threads.

Surprisingly, they *slowed down*!

JIM TRACKED DOWN THE CAUSE

It turned out that with more and more load on the database server, hence lots of threads, the database locking algorithm was discovering a lot of deadlocks.

Running the cycle detector, aborting all of those waiting threads, rolling back and then retrying – it all added up to huge overheads!

Jim showed that once this occurred, his databases slowed down

THE "FULL STORY"

He found that if you have a system with **t** threads or servers, and the system is trying to process **n** "simultaneous" operations (transactions), it could slow down as

$$O(n^3 t^5)$$

You added threads or servers to have your system handle more load

… but it slows down, dramatically!

… NOT WHAT WE WANTED!

People who buy a NUMA machine and run a program with more threads want *more* performance, not *less!*

Also, the situation Jim identified didn't arise instantly. It only showed up under heavy load. This made it hard to debug…

- A Heisen-performance-bug!
- Very bad news… Hard to find, impossible to fix!

WHAT DID JIM RECOMMEND?

He found ways to slice his big data sets into n distinct, independent chunks. He called this *sharding*.

Then he put each shard – each chunk of data – into its own database. He ran the n databases separately!

… like when fast-wc had a separate std::map for each thread.

SUMMARY

Deadlock is a risk when we have concurrent tasks (threads or processes) that share resources and use locking.

There are simple ways to avoid deadlock, but they aren't always practical. Ordered locking is a great choice, if feasible.

Complex options exist, but they can have high overheads.