

# 09-01

---

## 09 – Multi-Threaded Index Concurrency Control (CMU Databases Systems \_ Fall 2019)

00:15 – 00:17

Okay, guys let's get started

Ok, 孩儿们我们上课吧

00:19 – 00:23

again Thank You DJ drop tables always always always keep it down

感谢DJ Drop Table每节课所为我们做的事情

00:25 – 00:26

how's your mixtape going oh it's going

你的专辑如何了

00:27 – 00:31

we'll release it at the end of semester hopefully

我们希望能在学期末听到你的作品吧

00:31 – 00:33

okay and it's all it's all like DJ drop table beats

都是你打的拍吗?

00:34 – 00:36

oh nothing with you and like your own stuff

还是你啥也没干, 都是你的队友做的

#####

00:40 – 00:42

Okay alright, so let's get started

Ok, 我们开始上课吧

0.42-0.43

it's a beautiful day out

今天是美好的一天

0.43-0.51

and I think that's why the turnout here so low which sucks cuz my every lecture is all so much here ,but this one I like a lot too

我觉得这就是为什么今天的到座率有点低的原因, 这有点操蛋, 因为我每节课都干货满满

## ADMINISTRIVIA

**Project #1** is due Fri Sept 27<sup>th</sup> @ 11:59pm

**Homework #2** is due Mon Sept 30<sup>th</sup> @ 11:59pm

**Project #2** will be released Mon Sept 30<sup>th</sup>

00:52 – 00:55

Be before we get into the course material

但在我们开始上课之前

0.55–0.59

just to discuss real quickly what's on the schedule for you guys

快速讨论下我们接下来的规划

00:59 – 01:02

**Project #1** is due this Friday at midnight

Project #1这周五结束前要上交

1.02–1.05

and again you should submit that on great scope

你们应该把它们提交到GreatScope上

01:05 – 01:09

**Homework #2** is due on Monday at midnight also spindle grade scope

Homework 2是周一结束前截止，你们也要提交到GreatScope

01:09 – 01:13

So we'll send an announcement out on Piazza, but we've updated the PDF

So，我们会在Piazza上发布通知，但我们已经更新了PDF

01:14 – 01:17

So that you could drop the pictures in right into the the PDF

这样你们就可以将图片拖拽到PDF了

01:17 – 01:21

~~So you spitting that know~~, so we give you a template for draw IO

So，我们给你们提供了一个模板用来画I/O

01:22 – 01:25

So it's an online tool to go quickly edit and modify the templates for your answers

So，它是一个在线工具，它能让你快速编辑与修改你的答案模板

01:26 – 01:28

So should we know handwritten drawings

So，这上面支持手写

1.28–1.29

and no photographs of like drawings

但你们不能手写完拍照上传给到上面

1.29–1.31

that everything should be done digitally

所有东西都应该以电子形式完成

01:31 – 01:35

And then we'll be releasing **project #2** on this Monday as well

接着，我们也会在周一将Project #2放出来

1.35–1.40

and that'll ll do think two or three weeks in October, okay

它的任务周期大概是要花2到3周

01:40 – 01:43

So any high-level questions about the project or homework 2

So, 对于Project或者homework 2你们有任何高逼格的问题吗

01:47 – 01:49

Okay, so let's get in this

Ok, So 让我们上课吧

## OBSERVATION

We assumed that all the data structures that we have discussed so far are single-threaded.

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.

01:50 – 01:53

So the thing we need to talk about now is

So, 我们现在需要讨论的是

1.53–1.57

that we spent the last three classes talking about data structures

我们花了三节课左右来讨论数据结构

01:57 – 02:02

We'd spend down hash tables and spent two days on B+tree,radix trees and other tree data structures

我们已经讲了hash table, 然后又花了两天在B+ Tree, Radix Tree以及其他树形结构上面

02:03 – 02:07

So for the most part during this entire conversation when we talk about these data structures

So, 在这整节课的大部分时间中, 我们会讨论这些数据结构

02:08 – 02:13

We've assumed that they were only being accessed by a single thread

我们假设这些数据结构只能被一条线程而访问

02:14 – 02:17

But there was only one thread that could be reading and writing data to the data structure at a time

但只有一条线程能够在同一时间对该数据结构进行读写数据

但在同一时间只有一条线程能够对该数据结构进行读写数据

02:18 – 02:20

And that simplified the discussion

这也就降低了我们的讨论难度

2.20–2.24

and so that you just understand what's the core essence of how these data structures work

这样你们也就能理解这些数据结构工作方式的核心本质了

02:24 – 02:25

But in a real System

但在一个真正的系统中

2.25–2.30

we obviously don't want to just have a single thread be you know only accessing the data structure at a time

很明显，我们并不想同一时间只让一个线程去访问这个数据结构

02:31 – 02:32

We would allow multiple threads

我们想允许多个线程能在同一时间访问这些数据结构

2.32\*–2.35 。 。 。 。

because a modern CPUs there's a lot of CPU cores

因为在现代CPU中，它里面拥有大量的CPU Core

2.35–2.40

so therefore we can have multiple threads running queries and all updating our data structures

因此，我们可以通过多线程来执行查询，并更新我们的数据结构

02:40 – 02:46

But also don't allow this the high disk stalls due to you know or stalls due to having to go read read things from disk

同样，我们也不需要让cpu挂起等待从硬盘读取数据

2.46–2.50

because now if one thread is doing something and it reads a page that's not in memory

因为现在如果一条线程在做某些事情，比如读取一个不在内存中的page

02:50 – 02:53

It has to get stalled while the buffer pool manager brings that in

当buffer pool管理器将这个page放到buffer pool中时，这个线程就不得不停下来

2.53–2.55

and then we can let other threads keep running at the same time

那么，我们可以让其他线程在同一时间继续运行

02:56 – 2.59

So we're have a lot of threads running in our system

So，在我们的系统中运行着大量的线程

2.59–3.05

and we do this,because that maximizes parallelism or maximizes the reduces the latency for the queries you want to execute

我们这样做的原因是，因为这可以最大化并行能力，或者是最大程度上减少我们想要执行查询时的延迟

03:06 – 03:13

So for today we're now talked about up now we bring back multiple threads ,and want to update ,and access our data structure what do we need to do to protect ourselves

So，今天我们会去讨论多线程，我们想通过多线程来更新和访问我们的数据结构，我们该如何保证线程安全呢？

03:14 – 03:15

So let's say as it a quick aside

So，让我们先说一下

3.15–3.17

so everything that we'll talk about today is

So，今天我们所讲的一切东西

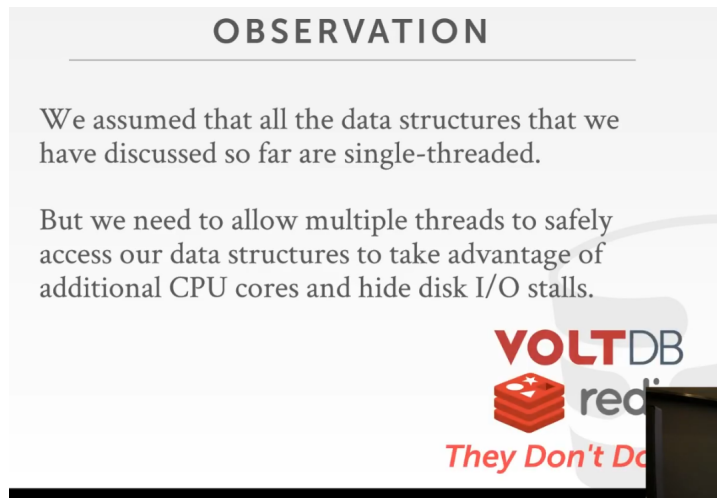
3.17–3.20

what how most database systems actually work

都是关于大部分数据库系统实际如何工作的

03:20 – 03:22

Most database systems that support multiple threads  
大部分数据库系统都支持多线程



3.22-3.26

will do the things that we're talking about today doing this latching stuff  
我们今天会去讨论使用latch锁这个事

03:26 - 03:28

There are some particular systems that actually don't do any of this  
有些特殊的数据库系统实际上并不会做这些事情

3.28-3.35

and only that single-threads has to access the data structures, and they still get really good performance

即使只有单条线程能访问该数据结构，但它们依然可以获得良好的性能  
HIDE DISK I/O STALLS.



03:35 - 03:38

So VOLTDB and Redis applied to 2 famous to do this——miscreants one set dips  
So, VOLTDB和Redis就是其中两个著名代表

03:38 - 03:39

So in case the Redis

So, 以Redis为例

3.39-3.42

Redis only runs in one thread, it's a one threaded engine

Redis只在一条线程中运行，它是一个单线程引擎

03:42 - 03:43

In VOLTDB it's a multi-threaded engine

VoltDB则是一个多线程引擎

3.43-3.49

but they partition the database in such a way that every B+tree can only be accessed by a single thread

但它们以某种方式将数据库进行分割，即每个B+ Tree只能由一条单个线程来进行访问

03:49 - 03:52

So you avoid all this latching stuff that we talked about today

So, 这样你就避免了使用latch的情况, 这也是我们今天要讨论的内容

3.52–3.54

and you get really great performance,

并且你也获得了非常优秀的性能

3.54–3.59

but obviously this means that it complicates scaling up to multiple cores or multiple machines

但很明显, 将它扩展到多核或多台机器上就会很复杂

03:59 – 04:03

But again we'll talk I'll just talk about these things later on in the semester

但我们会在这个学期稍后的时间里谈论这个

04:03 – 04:07

But the main idea now is that everybody pretty much does this things that we're talking about

但现在主流的思想就是, 所有数据库系统都有花大量精力在做我们所正在讨论的这些事情

## CONCURRENCY CONTROL

A **concurrency control** protocol is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.

A protocol's correctness criteria can vary:

→ **Logical Correctness:** Can I see the data that I am supposed to see?

→ **Physical Correctness:** Is the internal representation of the object sound?

04:09 – 04:14

So the way we're going to protect our data structures is through a concurrency protocol a concurrency scheme

So, 我们保护我们数据结构的方式就是通过一种并发协议或者并发方案来解决

04:14 – 04:22

And this is just the the method in which the database system guarantees the correctness of the data structure

这是数据库系统用来保证数据结构正确性的一种方式

4.22–4.27

by enforcing all the threads to access the data structures, and using a certain protocol or sort of certain way

即强制所有访问数据结构的线程都使用某种协议或者是某种方式

04:29 – 04:32

And so I'm putting the the word correct in quotes,

这里我给correct加了双引号

4.32–4.34

because that can mean about mean different things

因为这意味着一些不同的事情

04:34 – 04:38

And the kind of things were talking about they're accessing although we've been focused on data structures,

虽然我们现在所讨论的是访问数据结构

4.38–4.41

but it really could be for any shared object in the system

但这真的可以适用于系统中的任何共享对象

04:41 – 04:47

Right, it could be for tuple, could be for an index ,could be for the page table ,and the buffer pool,it doesn't matter

它也可以用于tuple, 所以, page table以及buffer pool, 这些都可以用到

04:48 – 04:55

So the two types of correctness we care about in concurrency control are **logical correctness** and **physical correctness**

在并发控制中我们所关心的两种正确性类型是逻辑正确性和物理正确性

04:55 – 04:58

So **logical correctness** would be like a high level thing that says

So, 逻辑正确性是一种高级层面的东西

4.58–5.01

if I'm accessing the data structure

如果我正在访问该数据结构

5.01–5.06

am I seeing the values or am I seeing the things that I expect to see

我是会看到值,还是会看到我希望看到的东西呢?

我希望看到的值是自己想看到的

05:06 – 05:07

So if I have a B+ tree index

So, 如果我有一个B+ Tree索引

5.07–5.09

I insert the key five

我将key 5 插入

5.09–5.12

my thread that thread comes back and reads key five right away,

我的线程会回过头来, 并立马读取key 5

5.12–5.13

it should see it

它应该可以看到它

5.13–5.16

right I should not get a should not get a false negative

它所得到的不应该是false或者negative

05:16 – 05:20

But that's a logical correctness thing that I'm seeing the things I that I expect to see

这就是所谓的逻辑正确性, 即我看到了我希望看到的东西

A protocol's correctness criteria can vary:

→ **Logical Correctness:** Can I see the data that I am supposed to see?

→ **Physical Correctness:** Is the internal representation of the object sound?

05:21 – 05:24

The thing that we're gonna care about in this class is **physical correctness**

我们这门课中所关心的是物理正确性

05:24 – 05:28

But how do we protect the internal representation of the data structure

但我们该如何保护数据结构的内部表示呢?

5.28-5.32

how it maintains pointers and references to other pages and keys and values,  
它该如何维护指针以及指向其他page的引用，还有key和value呢？

5.32-5.38

how do we make sure that as threads are reading writing this data that the integrity of  
the data structure is sound

我们该如何确保线程读写数据时，该数据结构的可靠性呢？

05:39 - 05:44

~~So it's able to be~~ we don't want the case where we're falling down traversing into the B+  
tree

So，当我们向下遍历B+ Tree时

5.44-5.47

and when we jump to the next node, we have a pointer to that

当我们跳到下一个节点的时候，我们需要一个指向它的指针

05:47 - 05:50

And then by the time we read the pointer figure out where we need to go

接着，当我们读取指针，以此来弄清楚我们需要往哪里走

05:50 - 05:51

And then then try to jump there,

然后，就试着跳到那个位置

5.51-5.54

somebody else modifies the data structure well

其他人也会去对该数据结构进行修改

5.54-5.58

now that pointer is pointing to a an invalid memory location,

现在，该指针指向了一个无效的内存位置

5.58-6.01

and we would get a Segmentation fault

现在，我们就会得到一个Segmentation fault（存储器区块错误，简称segfault）

06:01 - 06:02

So this is what we're trying to do today,

So，这就是我们今天所要试着解决的事情

6.02-6.06

we're trying to protect the internal data structure to allow multiple threads read and  
write to it

我们会试着保护内部数据结构，并允许多条线程对其进行读写数据

6.06-6.11

~~and that they still that~~ the data structure is behaving correctly

该数据结构依然能被正确使用

06:11 - 06:13

For the logical correctness

对于逻辑正确性来说

6.13-6.16

we'll worry about this more when we talk about transactions and concurrency control

当我们在谈论事务和并发控制时，我们会对此更为关心

06:16 - 06:19

All right, this is a whole another super interesting topic

这是另一个我们非常感兴趣的主题



6.19-6.24

but for today we say you know happy make sure that the data structures are thread safe  
但今天我们要做的，就是确保该数据结构是线程安全的



06:24 – 06:30

So we'll begin by talking about what is actually a latch they're a bit more detailed than we then we talked about so far, and how it's actually implemented

So，我们先会从什么是latch开始，这要比我们目前为止所讨论的内容来说，要更深入细节，并且我们还会谈论它实际是如何实现的

06:31 – 06:37

And then we'll start off with an easy case of actually doing thread safe hash tables using latches for those

接着，我们会去看一个简单的例子，即如何使用latch来保证hash table的线程安全

6.37-6.39

because that they're actually really simple to do

因为它们实际上非常简单

06:39 – 06:42

But then we'll spend most time talking about how to handle in B+tree

但之后我们会将大部分的时间用来讨论该如何在B+ Tree中进行这样的处理

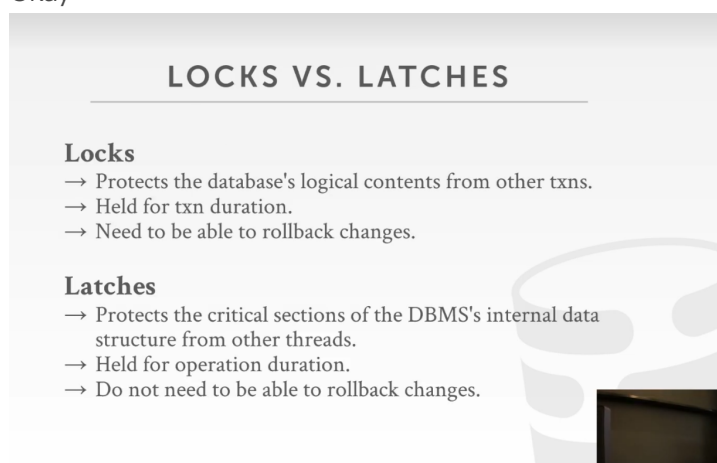
6.42-6.49

and what's talking how do leaf node scans ,and other optimizations again when we have multiple threads accessing things same time

然后，当我们使用多个线程同一时间访问这些东西时，我们会去讨论该如何对叶子节点进行扫描，以及一些其他优化

06:49 – 06:49

Okay



06:51 – 06:53

All right,so I showed this slide last time  
So, 我上次已经向你们展示过了这张幻灯片  
6.53-6.57

and I don't think everyone you know we only talked about very briefly  
我们上次讲它的时候, 只是大概讲了下  
6.57-6.58

and I don't think everyone absorbed its on it  
我并不觉得每个人都从中学到了什么  
6.58-7.02

I want to spend more time talking about the difference between **locks** and **latches**  
我想花更多的时间来讨论lock和latch之间的区别  
07:03 - 07:06

So in the database world where I live  
So, 在我所在的数据库世界中  
07:07 - 07:10

A **lock** is a higher level concept  
lock是一种更高级层面的概念  
7.10-7.14

that protects the logical contents of the database  
它保护了数据库中的逻辑内容  
07:14 - 07:20

So a logical content would be like a tuple or ,a set of tuples, or a table, a database  
So, 逻辑内容可以是一个tuple, 或tuple的集合, 或一张表, 或者是数据库  
07:21 -07:27

And we're having using these locks to protect these logical objects from other  
transactions that are running at the same time  
当同一时间有其他事务在运行时, 我们会使用这些lock来保护这些逻辑对象  
07:28 - 07:31

Like if I'm modifying something in a transaction  
比如, 如果我在一次事务中修改某些东西  
7.31-7.36

and so I don't want anybody else to modify that tuple at the same time that I am  
So, 我不想让其他人在同一时刻也去修改该tuple  
07:36 - 07:40

Right, you may for other reasons but for our purposes assume that we don't want that to  
happen  
但出于我们的目的, 我们并不想让这种事情发生  
07:41 - 07:46

So for these locks we're gonna hold them for the entire duration of the transaction  
So, 我们会在整个事务的执行期间持有这些lock  
7.46-7.47

again that's not entirely true  
再说一遍, 这并不是完全正确  
7.47-7.50

but again for our purposes just assume that's the case  
但出于我们的目的, 我们就以这种情况为例  
07:50 - 07:56 ! ! !

And then we need to be able to roll back any changes we make to the objects we modify  
if we hold the locks for them

然后，我们需要能够将这些对象回滚到任何操作前的状态，如果我们持有它们的锁的话

07:57 – 08:00

So if I'm trying to transfer money from my account to her account

So, 如果我试着将钱从我的账户转到她的账户

8.00–8.02

if I take the money out of my account

如果我将钱从我的账户取出

8.02–8.05

and then I crash before I put the money in her account,

在我将钱放入她的账户前，我遇上银行系统崩溃

8.05–8.08

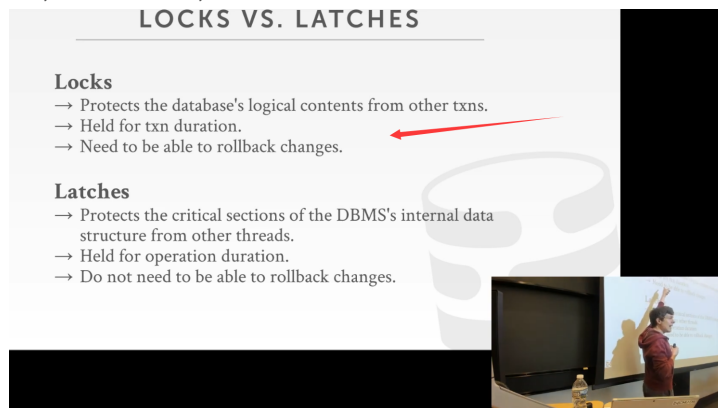
when I come back, I want to reverse that change I made to my tuple

当我这里系统恢复正常后，我想要撤销对该tuple所进行的修改

08:09 – 08:13

So these so in that means the database system is responsible for knowing how to roll back these changes

So, 这就意味着，数据库系统要负责对这些修改的回滚



08:14 – 08:15

So notice up here

So, 注意上面这里

8.15–8.18

I didn't say anything about threads or I'm talking my transactions

我并没有谈论任何关于线程的事情，我讨论的是事务

08:19 – 08:23

So a single transaction could be broken across multiple threads

So, 一个单个事务可能会被多个线程破坏

8.23–8.26

and they could all be updating the same tuple

这些线程可以更新同一个tuple（知秋注：如果没有锁的话，就出大事了）

08:26 – 08:27

That's okay, that's allowed

这种是Ok的，我们也允许

8.27–8.29

because the transaction holds the lock

因为在执行事务的时候，我们持有锁

8.29–8.32

it doesn't matter what thread that that's actually doing the modification

所以在实际执行过程中，不管哪个线程进行修改，都没问题

08:32 – 08:42

Where we get down to the low level constructs that we care about protecting the physical integrity of our data structures or the objects is **latches**

在低级层面，我们所关心的就是用来保护数据结构或对象的物理完整性的latch

08:42 – 08:44

So in the operating system world

So，在操作系统的世界中

8.44–8.46

they this is what they call locks or **mutexes**,

它们将latch叫做lock或者是mutex

8.46–8.50

in our world there's latches because we need to distinguish them from locks

在我们的世界中，我们将它叫做latch，因为我们需要将它们和lock区分开来

08:50 – 08:53

So latches are going to protect the critical sections of the database systems

So，latch会去保护数据库系统的关键部分

8.53–9.00

internal data structures from other threads that are reading writing into that data

structure or that object at the same time

即latch可以保护内部数据结构免受其他线程对该数据结构或对象同一时刻进行读写所带来的问题

09:00 – 09:07 ! ! ! ! !

So we're only hold latch for a short period just for the duration that were in the critical section to do whatever operation that we need to do

So，我们只会在一小段时间内持有latch，我们只会在对关键部分进行所需操作时持有这个latch

09:07 – 09:09

I want to update a page

比如，我想去更新一个page

9.09–9.13

I hold the latch on that page make the change then release the latch

在我对该page进行修改的时候，我会持有这个latch，修改完后，我会对它进行释放

09:13 – 09:15

We don't need to be able to roll back any changes here

这里我们不需要能够去回滚任何修改

9.15–9.21

because the operations we're trying to do are essentially meant to be atomic

因为我们所尝试要进行的操作，本质上来讲是原子性操作

09:21 – 09:23

So I hope I grabbed a latch from something

So，我希望我能抢到一个latch

9.23–9.24

I make whatever change I want

那我就可以做我想做的任何修改了

9.24–9.29

and then when I release the latch then the operations considered done,

接着，当我释放了这个latch后，那么这些操作就被认为是完成了

9.29–9.32

so all the changes are there

So，这样就执行了所有的修改

09:32 – 09:33

If I can't acquire the latch

如果我无法获取这个latch

9.33-9.35

then I'm not gonna do the operation anyway

那么我也就没法进行这些操作

9.35-9.38

so there's nothing to rollback

So, 这也就无须回滚任何东西了

09:38 - 09:40

So another way to think about this is

So, 另一种思考这个的方式是

LOCKS VS. LATCHES		
	<i>Locks</i>	<i>Latches</i>
<b>Separate...</b>	User transactions	Threads
<b>Protect...</b>	Database Contents	In-Memory Data Structures
<b>During...</b>	Entire Transactions	Critical Sections
<b>Modes...</b>	Shared, Exclusive, Update, Intention	Read, Write
<b>Deadlock</b>	Detection & Resolution	Avoidance
<b>...by...</b>	Waits-for, Timeout, Aborts	Coding Discipline
<b>Kept in...</b>	Lock Manager	Protected Data Stru

9.40-9.46

this great from the that that B+tree book I recommended a few lectures ago from Goetz Graefe

这个表是我前几节课提到那本B+ Tree相关的书里所提供的, 这本书是由Goetz Graefe编写的

09:46 - 09:50

We has this nice table that lays out again the distinction between **locks and latches**

这张表上列出了lock和latch之间的区别

09:51 - 09:54

So for locks we're gonna separate **user transactions** from each other

So, 在lock中, 我们会将用户的事务彼此之间分开

so, 对于lock来说, 我们会将彼此间的用户事务分开

09:54 - 09:59

And they're gonna be protecting the **database contents, tuples, tables** things like that lock能用来保护数据库内容, tuple, 表之类的东西

09:59 - 10:01

And we're gonna hold them for the **entire** duration of the **transaction**

在我们执行事务的整个期间内, 我们都得持有lock

10.01-10.06

there's gonna be a bunch of different lock types that we can help hold on these objects

这里有一些不同类型的lock, 我们可以在对对象进行修改时, 持有它们

10:06 - 10:08

Again, we'll cover this in a few more lectures

这些我们会花几节课的时间来对此介绍

10:09 - 10:12

And then when it comes time to actually dealing with deadlock

当真正应对死锁问题的时候

10.12-10.22

we're gonna rely on some external coordinator a lock manager or transaction manager to resolve any deadlock that could occur

我们会去依赖某些外部协调者，比如，lock管理器，事务管理器，以此来解决任何可能会发生的死锁问题

10:22 – 10:25

And the methods we can use are **waits-for timeout aborts** or some other things

我们可以去使用这些方法，比如：Waits-for, Timeout, Aborts或者其他方法

10.25–10.26

and what we'll focus on these later

我们稍后会关注下这些



10:27 – 10:28

What we care about is over here

我们所关心的是这里

10:29 – 10:31

We have these latches they're gonna protect **threads** from each other

我们通过这些latch用于这些线程彼此间共享变量的保护

10.31–10.35

for our **in-memory data structures**

对于我们那些内存型数据结构来说

10:35 – 10:37

We're gonna protect the **critical sections** inside these data structures

我们会去保护这些数据结构中的关键部分

10.37–10.41

there's only going to be two lock modes **read and write**

latch只有两种模式，即读锁和写锁

10:41 – 10:47

And the way we're going to avoid deadlocks is to us being good good programmers

我们避免死锁的方式是让我们成为优秀的程序员

通过避免死锁来让我们成为一个优秀的程序员

10.47–10.50

which is nice for databases good equals expensive right

对于数据库来说这很nice，但这等于要付出昂贵的代价

10:50 – 10:55

So it's up for us to make sure that we write high-quality code in our data structures to avoid deadlocks,

So，这就取决于我们，我们要确保我们在数据结构中编写高质量的代码，以此来避免死锁

10.55–11.01

because there is no external thing like a transaction manager or lock manager that's going to rescue us if we have a deadlock

因为我们没法通过外部的东西，比如事务管理器或者是lock管理器来帮我们恢复，如果我们遇上死锁问题的话

11:01 – 11:07

It's up for us to design and implement our data structure in such a way that deadlocks cannot occur

我们得设计并实现出那种并不会发生死锁的数据结构

11.07–11.10

and we'll see what that looks like later on

我们之后会看到这种锁

LOCKS VS. LATCHES		
Lecture 17	<b>Locks</b>	<b>Latches</b>
Separate...	User transactions	Threads
Protect...	Database Contents	In-Memory Data Structures
During...	Entire Transactions	Critical Sections
Modes...	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
...by...	Waits-for, Timeout, Aborts	Coding Discipline
Kept in...	Lock Manager	Protected Data Stru

11:10 – 11:12

So again our focus is on here

So, 再说一遍, 我们的重点是这样

11.12–11.18

we'll discuss all this lock stuff in **lecture 17** after the midterm

我们会在期中考试后的第17节课上讨论Lock

11:18 – 11:20

Again, I find all the super fast thing

再说一遍, 我所找到的这些东西, 用起来速度都非常快

11.20–11.28

but this is like one **but the** the black arts of database systems, if you can you know actually make this stuff work

但这就像是数据库系统中的黑科技, 如果你知道该怎么让它工作的话

# LATCH MODES

## Read Mode

- Multiple threads can read the same object at the same time.
- A thread can acquire the read latch if another thread has it in read mode.

## Write Mode

- Only one thread can access the object.
- A thread cannot acquire a write latch if another thread holds the latch in any mode.

	Read	Write
Read	✓	✗
Write	✗	✗

11:28 – 11:31

All right, so let's talk about the latch modes work for that we can have

So, 我们来谈下我们所能使用的latch模式

11:32 – 11:33

Again there's only two modes **read and write**

再说一遍, 这里只有两种模式, 即读模式和写模式

11:34 – 11:37

So the latch is being held in read mode

So, 在进行读模式的时候, 我们要持有latch

so, 我们所持有的latch是读模式时

11.37–11.42

then multiple threads are allowed to share that read latch

那么我们就允许多条线程在同一时间去读取同一个对象

11:42 – 11:44

Right, because again it's a read-only operation

再说一遍，因为这是一个只读操作

11:44–11:47

so I can have multiple threads read the data structure at the same time

So，这样我就可以在同一时间让多条线程读取该数据结构

11:47–11:51

there's no conflict, there's no integrity issues that could occur

~~这里就不会起冲突问题，没有不会发生完整性问题~~

这里不会产生冲突问题，没有写操作的发生

11:51 – 11:52

So they can all share that

So，它们可以共享这个数据结构

11:53 – 11:55

If I take out the latch and write mode

如果我拿到的是写模式的latch

11:55–11:58

then I can only that's an exclusive latch

这是一种独占型的latch

11:58–12:01

only one thread can hold that latch in that mode at a time

在这个模式下，一次只有一条线程能持有这个latch

12:01 – 12:03

So if I hold it write latch

So，如果我持有write latch

12:03–12:07

I'm making changes nobody else can read that object that I'm protecting until I finish

我就会对该对象进行修改操作，直到我完成操作前，没有人可以读取该对象

12:09 – 12:11

All right, the only two modes we care about

这就是我们所关心的两种模式

12:11–12:14

think of this is like again multiple threads we share this one

再思考下，这种情况下（读模式），我们可以让多条线程访问这个对象

12:14 12:16

this is this is an exclusive latch

在这种情况下（写模式），它就是一个独占型latch

12:17 – 12:20

All right, so let's talk actually how you implement a latch in a real system

So，现在我们来讨论下该如何在一个真正的系统中实现latch

## LATCH IMPLEMENTATIONS

### Approach #1: Blocking OS Mutex

- Simple to use
- Non-scalable (about 25ns per lock/unlock invocation)
- Example: `std::mutex`



12:21 – 12:25

So the first approach is **probably the one** you're most familiar with

So, 第一种方式可能是你们最为熟悉的一种

12.25–12.28

you know when you take any kind of systems course or operating system course

当你们在学习任何一门系统相关的课程时, 比如操作系统课程

12.28–12.31

it's a blocking operating system mutex and blocking OS mutex

你们会学到Blocking OS Mutex

12:32 – 12:33

So this is the simplest thing to use

So, 这是用起来最简单的东西

```
std::mutex m;  
:  
m.lock();  
// Do something special...  
m.unlock();
```

12.33–12.37

because it's sort of built into the language, like it like in C++

因为这是内置在语言中的东西, 就比如: C++

12:37 –12:39

The standard template library has this thing std::mutex

它里面的标准模板库就有std::mutex

12.39–12.41

and it's really simple to use

并且使用起来真的很简单

12.41–12.42

you just declare it

你只需对它进行声明

12.42–12.46

then you call lock do something what you know on your the object you're protecting with it

当你要对你的对象进行某些操作的时候, 你可以调用lock对它进行保护

12.46–12.48

,and then you call unlock, and you're done

接着, 你再调用unlock, 这样操作就完成了

12:50 – 12:54

Right, so does anybody know how this actually works in the operating system

So, 在座的有人知道这玩意是如何在操作系统中工作的吗?

12.54–12.56

in least some Linux

至少是在某些Linux系统中

12.56–12.57

how did the mutex like this work

mutex是如何工作的呢?

12.57–12.57

yes

请讲

12:58 – 13:00

He says Futex, what is a Futex

他说是使用了Futex, 那么Futex是什么?

13:02 – 13:02

What's that

能说清楚点吗?

13:06 – 13:12

He said well, so he said Futex he's correct in Linux, Futex stands for fast userspace mutex

Well, 他说的没错, 就是使用Futex, Futex指的是fast userspace mutex

13:12 – 13:13

The way it works is that

它的工作方式是

13.13–13.15

there is the in user space

它是在userspace中的

13.15–13.17

meaning in the address space of your process

也就是你的进程中的地址空间里面

13.17–13.23

there'll be a memory location that has you know a like a bit usually a byte also

它会占用一点内存地址, 比如1bit或者1byte左右

13:23 – 13:29

But I'll have a memory location that you can then try to do a compare and swap on to to acquire that that latch

但我会通过这个内存位置来尝试进行一次CAS操作, 以获取这个latch

13:30 – 13:31

But then what happens is

但接着发生的事情是

13.31–13.32

if you don't acquire it

如果你没能获取到它

13.32–13.36

then you fall back to the the slower default mutex

那么, 我们会退一步使用速度更慢且默认使用的mutex

13.36–13.38

where that goes down into the operating system

这是操作系统层面的东西

13:38 – 13:39

So the idea is

So, 这里的思路是

13.39–13.42

you do a quick compare and swap and in userspace

你在userspace中进行一次快速的CAS操作

13.42–13.43

if you acquire it you're done,

如果你获取到这个latch, 那就行

13.43-13.47

if you don't acquire it, then you fall down to OS which is gonna be slower

如果你没能拿到，那你就得去使用速度相对很慢的OS层面的mutex

13:47 - 13:48

Because what happens is

因为这里所发生的事情是

13.48-13.52

if you go down OS and sit on a mutex inside the kernel

如果你到OS层面并调用内核中的mutex

13.52-13.56

then the OS aha well I know you're blocked on this mutex and you can't get it

然后OS就表示，我知道你被这个mutex给阻塞了，你没法拿到这个latch（知秋注：并将你放到一个等待队列中，等待调度器调度）

13:56 - 13:59

So let me tell the scheduler to do schedule, so you don't actually run

So，让我告诉调度器来进行调度。这样你实际上就运行不了了

14:01 - 14:02

And the reason why this is expensive

之所以这样做代价昂贵的原因是

14.02-14.07

because now the OS has its own internal data structures that is protecting with latches

因为OS有它自己的内部数据结构，会使用latch来保护它们

14:07 - 14:11

So you've got to go update now the discussion table to say this this process of this thread can't run yet

so 你就会得知这个争抢失败的线程无法运行了

14:12 - 14:12

So he's correct

So，他说的没错

14.12-14.15

fast user-space mutex is will be fast

Futex的速度会很快

14.15-14.17

cuz that's just a spin latch we'll talk about the next slide

因为这是一种spin latch，这个我们在下一张幻灯片会讲

14:18 - 14:21

But he fall down to OS then then then you're screwed

但如果我们回退到OS层面，使用mutex

14:22 - 14:25

So this is another great example were like we were trying to avoid the OS much as possible,

So，这就是另一个很好的例子了，这里面我们会尽可能的避免使用OS层面的东西

14.25-14.28

for the first project you guys use this,because it's fine

在第一个project中，你们可以使用这个，因为这种做法并没有问题

14:28 - 14:29

But if you have a high contention system

但如果你使用的是一个高竞争系统

14.29-14.32

then everybody is going down to the OS and that's that's gonna be a problematic  
那么，所有东西都使用OS层面的mutex，那么这就很成问题

## LATCH IMPLEMENTATIONS

### Approach #2: Test-and-Set Spin Latch (TAS)

- Very efficient (single instruction to latch/unlatch)
- Non-scalable, not cache friendly
- Example: `std::atomic<T>`

14:34 – 14:39

So the alternative is to implement ourselves using a spin latch or **test-and-set spin latch** (TAS)

So，另一种备选方案就是由我们自己去实现，即使用一个spin latch或者是TAS（test-and-set，可以认为是CAS）spin latch

14:39 – 14:42

So this is extremely an extremely Efficient,

So，这种做法会非常高效

14:42–14:43

it's super fast

它的速度超级快

14:43–14:46

because on modern CPUs ,there's a single instruction

因为在我们的现代CPU中，它里面有一条指令

14:46–14:50

there's an instruction to do a single compare and swap on a memory address

使用这条指令可以在一个内存地址上进行单次CAS操作

14:50 – 14:53

I think it just like I check to see whether the value of this memory address is what I think it is

即检查这个内存地址上的值是否和我认为的值相等

14:54 – 14:54

And if it is

如果相等

14:54–14:57

then I'm allowed to change it to my new value

那么我就允许它将原来的值变为新的值

14:57 – 14:59

So think of like the latch is set to 0

So，假设这个latch要将值设置为0

14:59–15:00

I check to see whether it's 0

我会去检查这里是不是0

15:00–15:01

and if it is

如果它是的话

15:01–15:02

then I set the 1

那么我将它设置为1

15:02–15:04

and that means I've acquired a latch

这就意味着我已经拿到了latch

15:04 – 15:07

And you can do that a modern CPUs and single instruction

你们可以通过现代CPU中的单条指令来完成

15:08 – 15:13

Right, you don't have to have you don't the write C code like if this then that ,it does it all for you

你无须去编写这样的C代码，比如if then这样的语句，这条指令会帮你做这样的事情

```
std::atomic_flag latch;  
:  
while (latch.test_and_set(...)) {  
    // Retry? Yield? Abort?  
}
```

15:14 – 15:21

So the way even implement this in C++ is that you had this atomic keyword which is templated ,you can put whatever you want there

你可以在C++中这样实现，使用这个atomic关键字，它是一种模板，你可以将它放在你想要的任意位置

15:21 – 15:24

But they have a shortcut for you called atomic flag

但它们使用了一个快捷方式，即atomic\_flag

→ Example: `std::atomic<bool>`

```
std::atomic<bool>  
std::atomic_flag latch;  
:  
while (latch.test_and_set(...)) {  
    // Retry? Yield? Abort?  
}
```

15:24–15:27

which is just an alias for atomic<bool>

它其实是atomic<bool>的别名

15:27 – 15:30

And so inside this now it will when we want to acquire this latch

So，在这段代码里面，当我们想去获取这个latch时

15:30 – 15:32

We have to have this while loop

我们必须使用这个while循环

15:32–15:35

that says test-and-set the latch

在它的条件判断部分有一个latch.test\_and\_set()

15:35–15:36

if I acquire it

如果我获取到这个latch

15:36–15:38

then I jump out of the the while loop

那我就跳出这个while循环

15:38–15:39

because I hold the Latch

因为我拿到了这个latch

15:39 – 15:41

If I don't

如果我没能拿到这个latch

15:41–15:42

fall into the while loop

那我就得进入这个while循环

15:42\*–15:46

and now it has some logic to figure out what should we do

并通过一些逻辑来弄清楚我们该做什么

15:46 – 15:50

The simplest thing is just say all right let me just retry again loop back around and keep trying it

So, 最简单的做法就是我们重新试着获取这个latch, 一直尝试去获取它就行

15:51 – 15:55

Right, the problem with that is though that's just me burning out your CPU you're not burning out literally

这种问题在于你会去燃尽你的CPU, 虽然并不是字面上燃尽的意思

15:55 – 15:57

But you just burning cycles and your CPU,

但这会不停的循环, 并且占用你的CPU

15:57–16:00

because you just keep trying to test that set test and set test the set

因为你会一直不断地尝试进行test-and-set

16:00–16:00

and it's always gonna fail

但这始终失败

16:00–16:03

and you keep spinning around and in this infinite loop

然后你就会一直在这个无限循环中自选等待

16:04 – 16:06

So the OS thinks you're actually doing useful work

So, 实际上OS会认为你在做些有用功

16:06–16:08

because it doesn't know what instructions you're executing

因为它并不知道你在执行什么指令

16:08 – 16:11

So it says you keep executing instructions let me keep scheduling you

~~So, 它表示, 你先继续执行这些指令, 我会为你继续调度~~

So, 它表示, 你要一直执行这个指令, 让我持续给你传功吧!

16:11–16:13

and you're to spike the CPU

这样, CPU的使用率就会激增

16:14 – 16:17

So this test and set thing is the same thing he said before about the fast user mutex,

So, 这就是test-and-set了, 它和之前他所说的Futex是一回事

16:17–16:23

this is the same thing the OS provides you in the Linux standard or the std::mutex on Linux

这和OS所提供给你的效果是一回事，比如Linux中的std::mutex所提供的锁效果一样（但实现形式不同，并不会一直无须循环进行TAS）

16:24 – 16:24

But maybe I don't want to burn my cycles

但可能我并不想去一直去榨干我的CPU Cycle

16:26 – 16:29

But he's keep retrying, maybe I want to yield back to the OS,

但它会不断重试，现在我可能就想回到OS层面来做些事情

16:29–16:32

let it schedule some other thread

让OS对其他线程进行调度

16:32–16:34

or maybe I try a thousand times

或者我尝试获取了1000次latch

16:34–16:37

and I'm saying I'm not gonna get this ,and I just up abort

如果我还没拿到这个latch，那就会进行中断

16:37 – 16:39

So this is a good example

So，这就是一个很好的例子

16:39–16:41

of where we as the database systems developer

我们作为数据库系统开发人员

16:41–16:47

we can be smart or we can tune the our implementation

我们非常聪明，我们可以对我们的实现进行调优

16:47–16:52

however using latches and our data structures to be mindful

但在使用latch和我们的数据结构时要注意

16:52–16:55

try to accommodate what we think the workloads gonna look like

不管我做什么操作，latch的获取与释放都应该很快

试着去想下我们的线程在TAS失败后会遇到的工作情况

16:55 – 17:00

~~If I think that this latch has to be like~~ whatever the operation I'm doing the latches to be super fast

不管我做什么操作，latch的获取与释放都应该很快

17:00–17:02

then it's probably faster for me to just keep retrying

那么对等待的线程来说进行重试的速度就可能很快

17:02–17:04

because whoever holds the latch will give it up real quickly

因为不管谁拿着latch，它都会很快放弃这个latch

17:04 – 17:06

But if I think the operation is going to be super long

但如果我觉得操作要花的时间太长了

17:06–17:12

then maybe I want to yield or for some amount of time or eventually abort

那么我可能会对线程进行yield操作，让其他的线程先执行，或者就直接中断操作

17:12 – 17:13

We can't do this in the blocking OS mutex

我们没法在blocking OS mutex中做这个

17.13–17.15

soon as we try to get it ,we can't get it

即当我们试着去获取锁的时候，我们没有办法让线程做让出cpu资源这个事儿（yield操作）

17.15–17.18

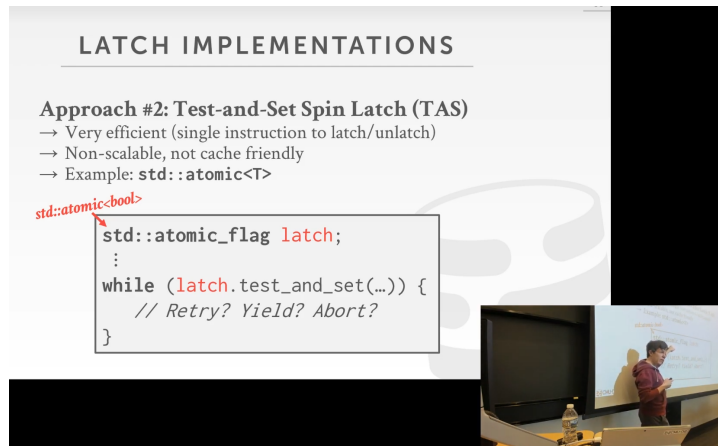
the OS takes over ,and we're blocked

OS就会去接手，然后我们就阻塞了

17:18 – 17:18

yes

请问



17:21 – 17:23

The questions what is this

你的问题是这个吗？

17:25 –17:27

this ,oh this

Oh, 这个啊

17:28 – 17:31

yeah like the primers would be like it's compare and swap

这里其实就是一个CAS操作

17.31–17.37

it says at this memory address check to see whether the value is this, like passing a zero

它表示，在这个内存位置，检查下该值是否是这个，比如这里传入一个0

17:37 – 17:39

If it if it equals zero, then set it to one

如果这里的值等于0，那么就将这个值设置为1

17:40 – 17:42

Right, and then there's different there's different API,

这里有些不同的API

17.42–17.44

sometimes you'll get back the old value

有时你会拿到老的值

17.44–17.45

you'll get back a true whether it's succeed

不管它CAS有没有成功，你拿到的都是true

17.45–17.47

there's a bunch different things



这里涉及到很多不同的东西

17:47 – 17:52

And then they have they have test and sets for you know for all the different types you could you could be based on

接着，这里的test\_and\_set()方法，你可以用于各种类型的数据

17:55 –17:56

So again the main takeaway here is

So，这里的主要要点是

17.56–17.59

that again we we in the database system can do a better job than the OS,

再说一遍，我们在数据库系统中所做的可以比OS给我们所提供的要来得更好

17.59–18.02

because we would know in what context we be using this latch

因为我们知道在哪个上下文中，我们会去使用latch

18:05 – 18:07

So for these two examples

So，在这两个例子中

18.07–18.10

though the latch has just been you know do I hold it or not

我是否持有这个latch呢？

18:11 –18:12

as I said before

正如我之前所说

18.12–18.14

we have different modes


我们有不同的锁模式

## LATCH IMPLEMENTATIONS

### Approach #3: Reader-Writer Latch

- Allows for concurrent readers
- Must manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks

**Latch**

	
read	write
 =0	 =0
 =0	 =0

18.14–18.20

so we need a reader writer latch that can support we have these different modes

So，我们需要一个reader-writer latch来支持这些不同的模式

18:20 – 18:27

the way we basically do this and we build on top of whatever our basic latching primitive we have either the spin latch or the POSIX mutex

简单来讲，我们是通过在基础的latch原语之上构建出spin latch或者POSIX mutex这种东西

18:27 – 18:33

and then we manage different queues to keep track of how many threads are waiting acquired to different types of latches

然后我们通过管理不同的队列来跟踪不同类型的latch有哪些线程在等待获取

18:34 – 18:36

right so it may be just mentioned some counters to say

So, 它这里面可能会使用一些计数器

18.36–18.39

here's the number of threads that help hold the Latch, in this mode

计数器会表示, 这里是持有该模式latch的线程数量

18.39–18.41

here's the number of threads that are waiting for it

这里是等待该latch的线程数量

18:41 – 18:45

so if a read thread shows up and says I want to get the read latch,

So, 如果这里有一个读线程, 它表示它想去获取read latch

18.45–18.46

well I look over here and say

Well, 我看了下这里, 并表示

18.46–18.50

nobody nobody holds the right latch and nobody is waiting for it

没有人持有这个latch, 也没有人正在等待获取它

18:50 – 18:52

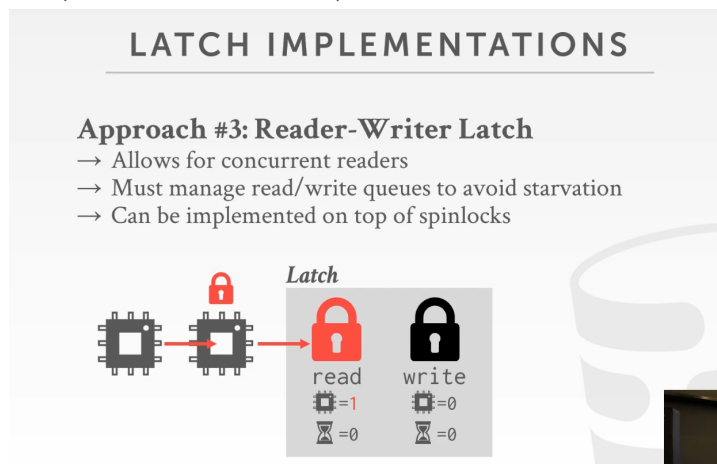
so I go ahead and and hand it out

So, 我把这个read latch分发给这个线程

18.52–18.58

and I update my counter to say I have one thread that holds this latch

接着, 我更新下我的counter, 并表示我有一条持有这个latch的线程



18:58 – 19:00

another thread comes along

这时又来了另一条线程

19.00–19.01

and once also quite a Reed latches

它也想获取一个read latch

19.01–19.04

again Reed latches are compatible or making me shared

再说一遍, 我们能够共享read latch