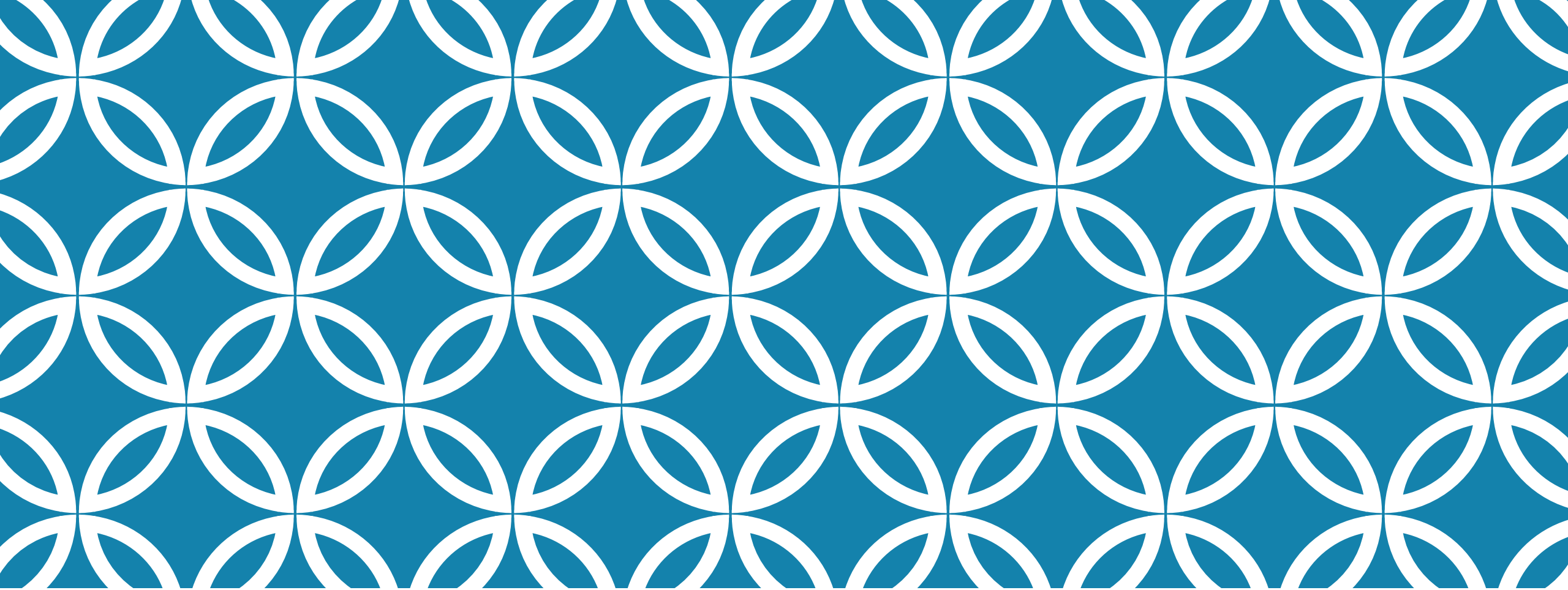


DEADLOCKS, LIVELOCKS, AND HOW TO AVOID THEM

Professor Ken Birman

CS4414/5416 Lecture 10

IDEA MAP FOR TODAY'S 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.

MORE EXAMPLES

```
…
std::unique_lock plock(mtx);
if(nfull == LEN) { release lock; wait; reacquire lock; }
…
```

Right here, before wait, context switch could occur

A is the only baker at Wide Awake Bakery. After baking a new batch of cinnamon raisin swirls, A unlocks and opens the display. Seeing that it is full, A releases the mutex and waits.

B is a customer. The instant A releases mutex, B grabs the lock and buys buys everything, then notifies the wait condition: "**case_not_full**" and releases the lock. But all of this happened while A was about to start waiting. The notify occurs first, so it does nothing.

Now A is waiting on **case_not_full**. Other customers that show up wait on **case_not_empty**.

Everyone is waiting. Nobody ever makes progress again! A deadlock…

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, and usually will deadlock silently. **A waits for A!**

Use a recursive mutex if you require recursive locking.

This kind of mutex is slower, but will count how many times the thread locked it.

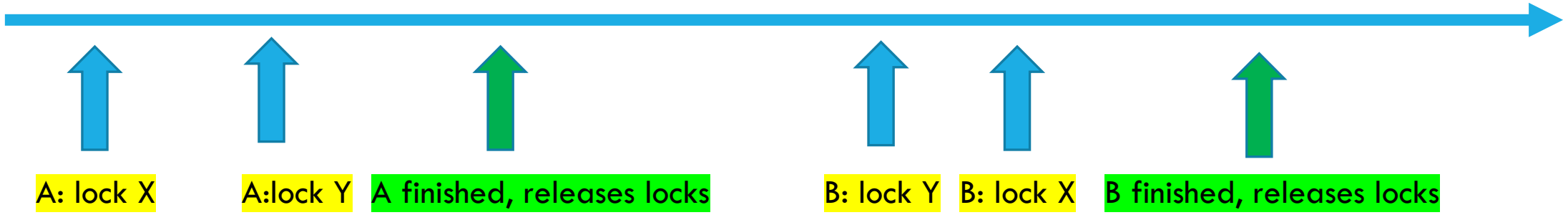
MORE EXAMPLES

A and B lock X and Y, *but not in the same order.*

Sometimes this can cause a deadlock… other times they manage to get away with it.

Examples on next slide.

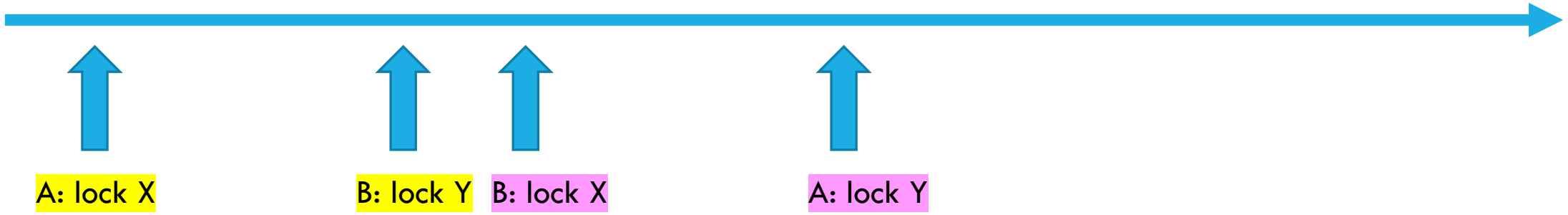
AS A TIMELINE PICTURE THE GOOD CASE



In this run, A and B got lucky. It was a race, but A won and got both locks, finished what it was doing, then released them.

B then runs, gets both locks, then releases them too.

AS A TIMELINE PICTURE: THE DEADLOCK



Trouble! Here, B grabbed the lock on Y while A was doing other stuff (but holding a lock on X). Now B wants a lock on X and A wants a lock on Y.

They get stuck: a deadlock!

COMMON HACK – BUT A MISTAKE!

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. The idea: sooner or later things should work…

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: **livelock.**

BETTER: LET THE PROGRAM GET INTO A DEADLOCK, THEN DEBUG THE ISSUE

Without knowing about how mutex is implemented you can't tell which thread is holding a lock.

But gdb can show you!

It can report the lightweight process "id" currently holding a lock

```
(gdb) thread 2
[Switching to thread 2 (Thread 0xb6d74b90 (LWP 22026))]#0  0xb771f424 in __kernel_vsyscall
()
(gdb) bt
#0  0xb771f424 in __kernel_vsyscall ()
#1  0xb76fec99 in __lll_lock_wait () from /lib/i686/cmov/libpthread.so.0
#2  0xb76fa0c4 in _L_lock_89 () from /lib/i686/cmov/libpthread.so.0
#3  0xb76f99f2 in pthread_mutex_lock () from /lib/i686/cmov/libpthread.so.0
#4  0x080484a6 in thread (x=0x0) at mutex_owner.c:8
#5  0xb76f84c0 in start_thread () from /lib/i686/cmov/libpthread.so.0
#6  0xb767784e in clone () from /lib/i686/cmov/libc.so.6
(gdb) up 4
#4  0x080484a6 in thread (x=0x0) at mutex_owner.c:8
8               pthread_mutex_lock(&mutex);
(gdb) print mutex.__data.__owner
$1 = 22025
(gdb)
```

LWP is a form of unique pthread id. The same post explains how to find the corresponding thread

C++ - Is it possible to determine the thread holding a mutex? - Stack Overflow

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 "busy wait"

For example 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.

HOW TO AVOID DEADLOCKS?

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

HOW TO AVOID DEADLOCKS?

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

Cool fact: std::scoped_lock will do this automatically if you list multiple mutexes in a single call.

But you do need to list all the mutexes you might use.

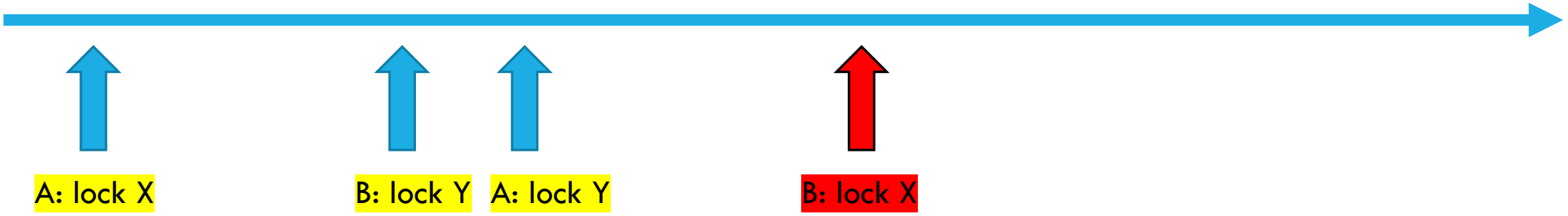
HOW TO AVOID DEADLOCKS?

More common is to lock in order by hand, with the sequence in mind.

Example: Recall A and B with X and Y. Use *alphabetic ordering*

- 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.
- **Nothing will check your logic!** But if you code this correctly, it works

AS A TIMELINE PICTURE



B shouldn't be trying to try to lock X while still holding a lock on Y,if you are using an ordered locking rule.

X is alphabetically smaller than Y, and B locked Y earlier. But C++ won't will throw an exception here. Your code didn't follow the rule… and nothing was checking

SO… USE ORDERED LOCKING! BUT IT CAN BE IMPRACTICAL

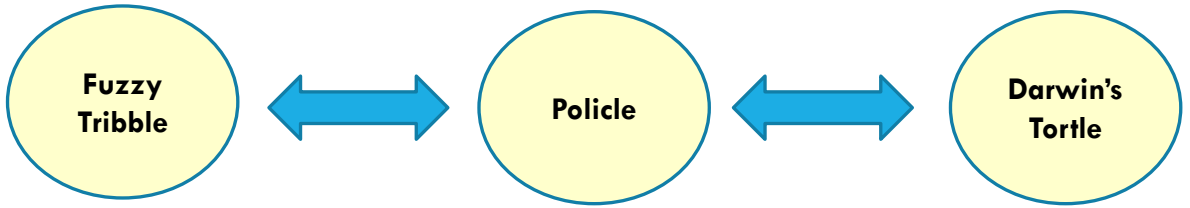
There are many applications that learn what they must lock one item at a time, in some order they cannot predict.

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

… now it is too late!

EXAMPLE: UNPREDICTABLE LOCK ORDER

For example, this could arise in a for loop. Maybe B is scanning a std::list<Species*>, and needs a lock on each Species.



The std::list isn't sorted by Species.name. The lock rule requires locks in Species-name sort order. B locks Fuzzy Tribble and Policle but now can't lock Ballard's Hooting Crane

WHAT IF IT TRIES?

This is a rule *you* would impose on yourself

If you don't respect your own design, that would be a bug in your code. C++ itself won't enforce this rule.

It definitely is possible to "wrap" locks in a way that would track locking and detect cyclic wait, but this isn't standard in C++

… EVEN SO, ORDERED LOCKING 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 ordered locking.

Many programs 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.

Assuming that the conditions giving rise to 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. That was what we showed you on slides 8, 9

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.

CYCLE DETECTION IS A COMMON DEADLOCK DETECTING METHOD

Many applications have self-checks for deadlocks (database systems, especially)

Periodically they build a graph of “who is waiting for whom”

Deadlocks are evident as cycles in these graphs

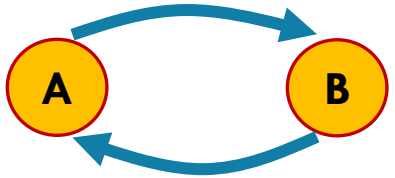
HOW TO CONTINUOUSLY CHECK FOR DEADLOCK

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 ALGORITHM?



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.

THE SAME METHOD CAN ALSO DETECT UNORDERED LOCKING

Because it tracks who is waiting for whom using a graph, it also knows if A had a lock on Y before it tries to get a lock on X

Now you can throw a "lock order exception" and debug the issue

Ken has done this in a very elaborate system with a lot of locking, as a debugging aide. He used a "const" **if** statement to disable this whole wrapper when shipping the product so it didn't slow it down at all.

PRIORITY INVERSIONS

In some systems, threads are given different scheduler priorities

- 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

Once we create our wrapped locking tool for deadlock detection, 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 (at priority p) holds a mutex on X.

B (with priority p' > p) wants a lock on X. B's lock request can also "bump" A to priority p' temporarily. The wrapper would also restore A to priority p later, when A releases the lock on X.

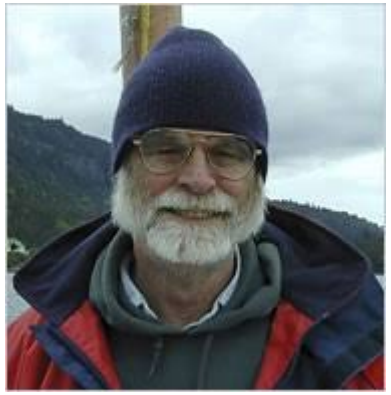
NONE OF THESE IS CHEAP…

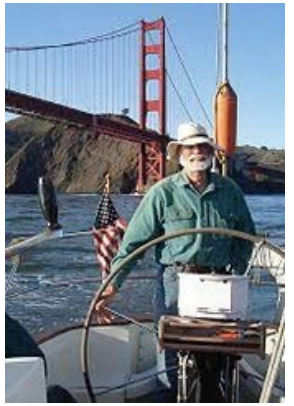
Recall our discussion of C++ versus Java and Python.

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

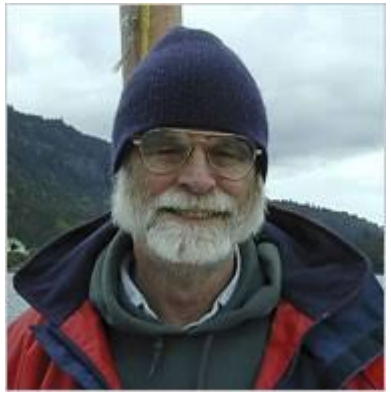


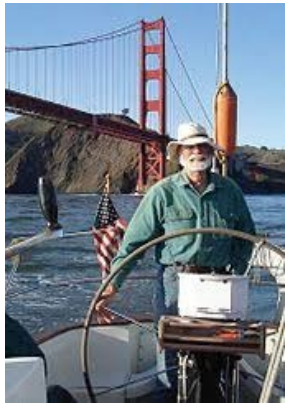


Jim Gray, a Turing Award winner, was a big player in inventing databases and “transactions”. He worked at Microsoft

Jim’s focus for much of his career was on making it easier to create really big databases and to access them from programming languages like C++ (or C#, Java, Python, whatever)

JIM GRAY'S STUDY





In the 1990's, databases were used for storing all forms of data. They use the kinds of ideas just listed (deadlock detection, priority tracking, rollback and retry, etc)

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*!

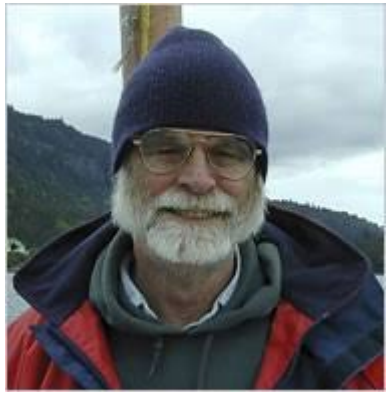
JIM TRACKED DOWN THE CAUSE

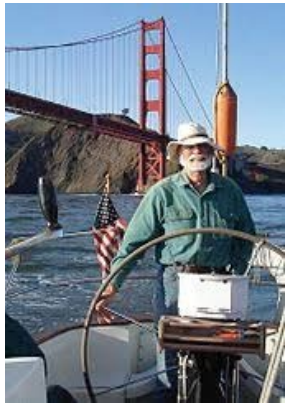
It turned out that with more and more load on the database server, hence lots of threads, the database locking algorithm was discovering a lot of deadlocks.

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"





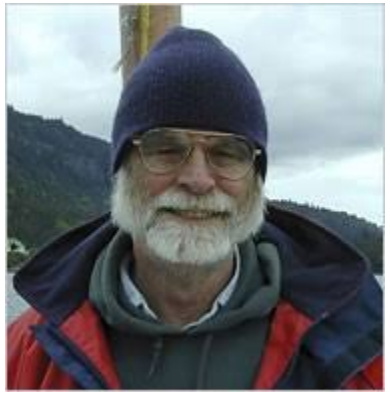
He found that if you have a system with **n** servers (or using **n** cores), and the system is trying to process **t** "simultaneous" transactions (transactions), it could slow down as

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

You used more cores servers to have your system handle more concurrent threads or transactions

… but it slows down, dramatically!

… NOT WHAT OWNERS EXPECTED!

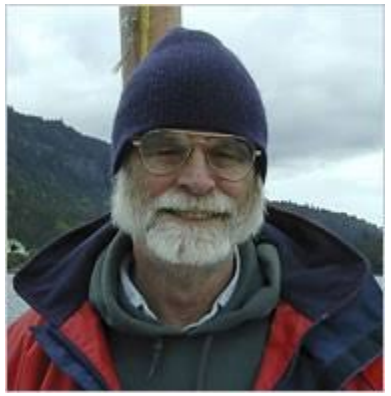


People who buy a NUMA machine and run a program with more threads want *more* performance, not *less!*

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!

EXAMPLE?



That isn't going to work

Suppose that your boss wants to increate capacity for a RAG ML system with a dynamically updated database.

A decision is made to double the number of servers. Over time the workload doubles too: n becomes 2n, and t becomes 2t.

… plug in $(2n)^3(2t)^5 =$ **old cost** * **512**

… Jim is saying the overheads will soar by 2^8

Yikes! We are spending twice as much but the system slowed down by 512x!

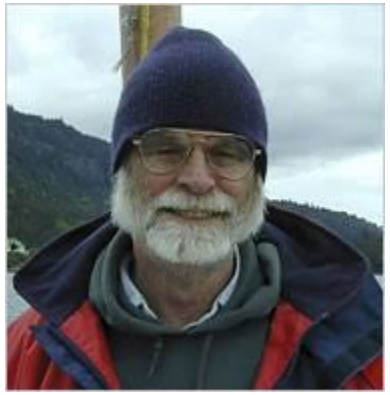
HOW DO YOU EXPLAIN THIS?

With the new workload, more deadlocks occurred. Like downtown Ithaca with more traffic: we get more jams.

Each caused a rollback and retry… some of *th*ose transactions deadlock too!

A form of exponential traffic jam!

WHAT DID JIM RECOMMEND?



He found ways to slice his big data sets into n distinct, independent **shards** (chunks).

With a sharding approach, we run all n databases separately! The rate of abort/retry drops by a factor of n^3

It works in many cloud settings and not just for databases.

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.

SUMMARY

Livelock is a form of deadlock in which threads or processes are active but no progress is occurring.

Often associated with some form of “busy wait” loop.

Deadlock avoidance mechanisms often can prevent livelocks, too

SELF-TEST

Remind yourself what the four conditions for deadlock were

Which condition can never occur with ordered locking? Why do you accept this "claim"?

Are there situations in real life (not in computing), where human beings use ordering to avoid deadlocks?

SELF-TEST

Have you ever seen a real-world (non-computing) example of a livelock?

What do people do to avoid these? Is it the same as a deadlock or different?

HOW HARRY AND SALLY BROKE UP

Harry and Sally often meet in the coffee shop or the cafeteria.

One important time, they want to meet, but Harry isn't sure Sally will have read his last email, which said the park is really beautiful today. Should he just go there? Or go to the coffee shop like usual?

Sally knows Harry is the nervous type, and she actually replied "ok, see yah!" but now *she* doesn't know if he read it…

HOW HARRY AND SALLY BROKE UP

H: How about meeting in the park instead of the coffee place?

S: The leaves are incredible now… I would love to!

H: Ok, in the park then

S: Yup, the park

… (repeats for a while)

S: Why did you go to the coffee shop? We agreed on the park!

H: No, we were still discussing it. We never really agreed

THIS IS A LIVELOCK!

Goal: K^*(park)
Prior: K^*(coffee shop)

The two processes (Harry and Sally) keep sending emails but never actually can deduce that they have agreed on the park.

In this particular case, Harry wanted to know that Sally knows that Harry knows …. (for all iteration values).

We can formalize this as a kind of "epistemic knowledge equation," shown at the top right corner. Sally wants the same thing.

LEARNING MORE?

In fact there are famous papers by Joe Halpern and his student Yoram Moses on problems like this

They show how reasoning about sequences of events (about time) can be incredibly confusing and unintuitive

Harry and Sally in an infinite exchange of emails is an example. Read about "knowledge and common knowledge" to learn more. The "muddy children" problem used in the introduction is classic.

THOUGHT QUESTIONS

As humans, do we reason in a purely logical way? Or do we engage in the kind of endless email exchanges Harry and Sally did?

There is a kind of knowledge asymmetry in an email back-and-forth dialog. Explain why. Can it be avoided?

Suppose Harry and Sally have reliable watches, and check email at least hourly. Does knowing this help them agree on a rendezvous?

TRAFFIC CIRCLES

Think of a traffic circle. Normally, entering cars yield, but there are some traffic circles in Paris where traffic lights cause cars in the circle to yield to cars entering the circle.

What does our theory of deadlocks tell us about this scenario?