DEADLOCKS, LIVELOCKS, AND HOW TO AVOID THEM

Professor Ken Birman
CS4414 Lecture 17

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

We only have some critical section

A locks it, but now needs to wait for some condition. The developer didn't use the monitor pattern, and instead drops the lock and then waits on some other form of mutex

But B ran as soon as the lock was released, and by the time A waits, the condition A was waiting for is already true.

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 <u>*are*</u> 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.

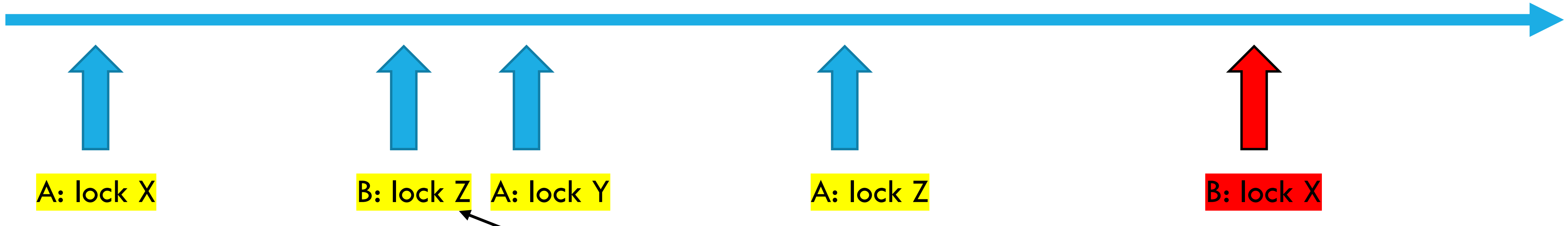
HOW TO AVOID DEADLOCKS?

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. 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.

AS A TIMELINE PICTURE



B isn't permitted to lock X under an ordered locking rule. X is alphabetically smaller than Z, and B locked Z earlier

… THIS RULE CAN BE IMPRACTICAL

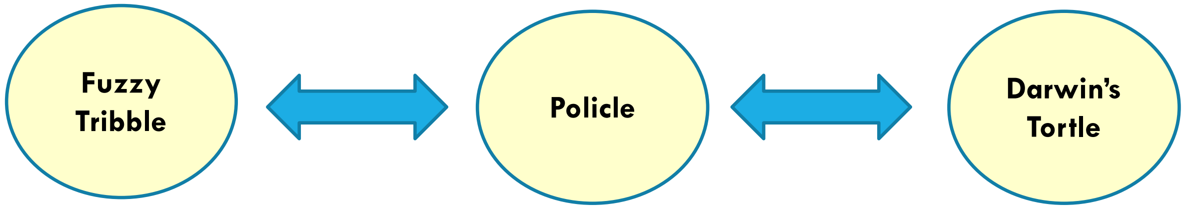
There are many applications that learn what they must lock one item at a time, in some order they cannot predict.

So in such a situation, B didn't know it would need a lock on X at the time it locked 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<Animal*>, and needs a lock on each Animal.



The std::list isn't sorted by Animal.name. The lock rule requires locks in Animal-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 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

"Okay — This is your last chance to back out."

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

"The system crashed. You'll have to go back and live it again."

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.

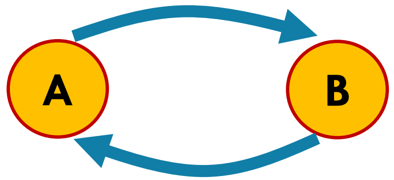
HOW TO BUILD A DEADLOCK DETECTOR

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.

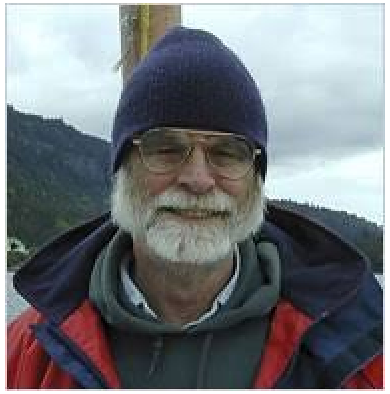
NONE OF THESE IS CHEAP…

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!

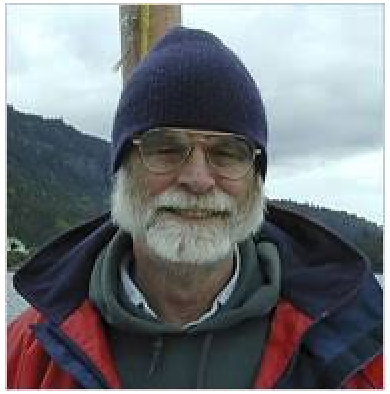
JIM GRAY



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

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*!

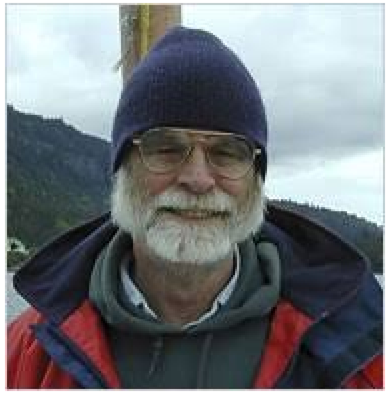
JIM TRACKED DOWN THE CAUSE

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"



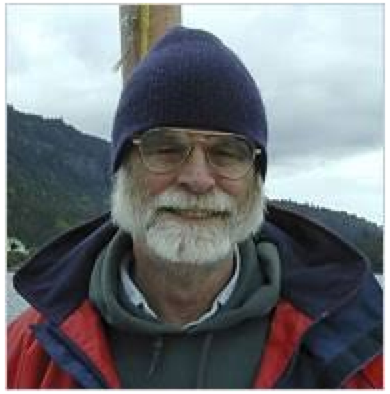
He found that if you have a system with **n** servers (or using **n** cores), and the system is trying to process **t** "simultaneous" transactions (transactions), it could slow down as

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

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

… but it slows down, dramatically!

SIDEBAR



Jim didn't specifically think about **n** being cores, and **t** perhaps being threads that access shared variables and locks.

But this is a reasonable generalization and in fact holds, if the threads exhibit a pattern of synchronization similar to the one he examined in his famous paper, where a transaction touches variables randomly, using an i.i.d. access pattern.

… 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*.

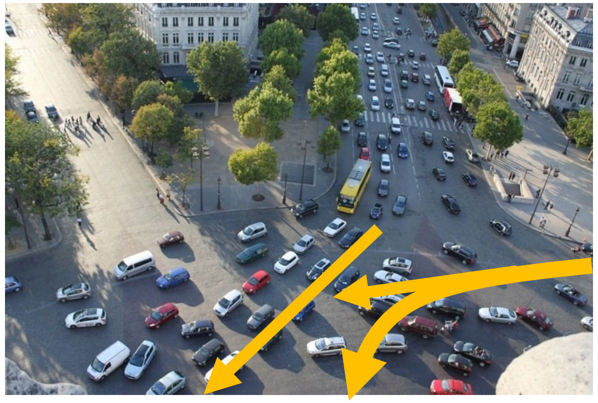
Then he put each shard – each chunk of data – into its own database. He ran the n databases separately!

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

TRUE THOUGHT PROBLEM

Ken was on sabbatical in Paris in 1995-1996

There are two traffic circles in France in which priority favors allowing cars to *enter* the circle over exit.

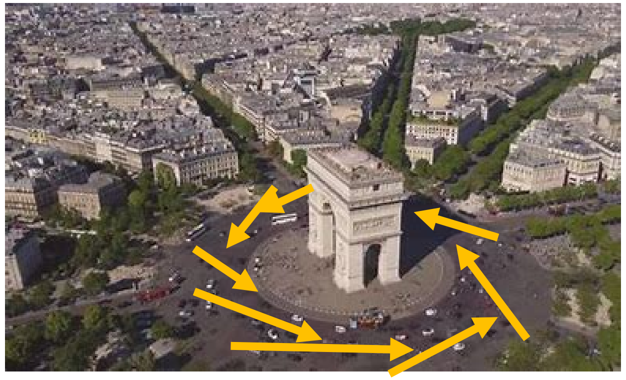


Deadlock occurs! Would "priority to the left" have the same risk?

VISUALIZING THE PRIORITÉ-À-DROITE CYCLE

Many streets. Traffic flows in counter-clockwise direction

Streets are mostly two-way



Core problem: some cars just want to leave, but others continue around. **But any entering car gets to go first**

PRIORITY TO THE LEFT IS THE MODERN RULE

Most of the "drives on the right hand side" world uses the priority-to-the-left rule.

For a traffic circle, this means cars in the circle have priority over cars wanting to enter. "Drains" traffic out.

But these two traffic circles, and also the whole country of Belgium, use priorite-a-droite and deadlock.

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