Toward General Deadlock of Locks Prediction in Lockdep

Du, Yuyang

滴滴金融

Outline

- Introduce deadlock
- Demystify lockdep
- Read-write locks
- General deadlock of locks prediction algorithm

Deadlock

（图片来自网络）

Every car is alive but cannot make forward progress forever!

This happens on multi-threaded or distributed-system programs too.

Why deadlock happens?

（图片来自网络）

The story of three monks no water.

A deadlock is caused by circular waiting relationship among the actors.

Deadlocks are bad

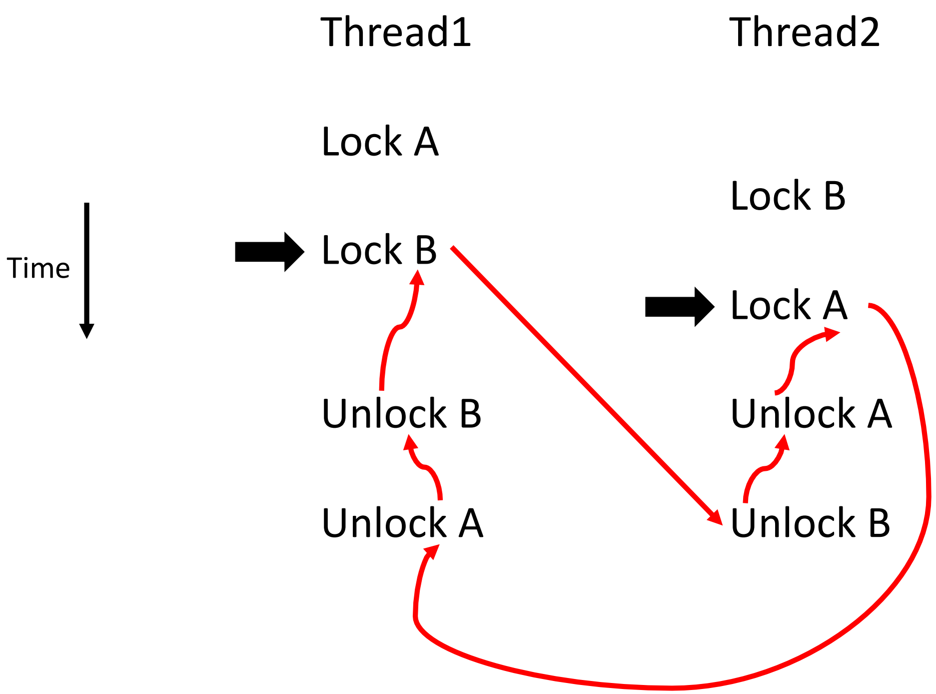
- If happened, it is devastating
- Concrete vs. potential deadlocks
  - E.g., timing conditions are met
- If not so far, it will sometime according to Murphy's Law
  - "Anything that can go wrong will go wrong"
- Deadlock is notoriously hard to debug

What can be done about deadlocks?

- Detection
  - After concrete deadlocks happened
- Prevention
  - Before concrete deadlocks happen
- Prediction
  - Find potential deadlocks
  - A mechanism to characterize the locking behavior of workloads
  - Then based on the behavior report potential deadlocks

Deadlock of locks

Example:



What a long circle! So complicated!

Waitings:

- Thread1 Lock B -> Thread2 Unlock B
- Thread2 Unlock B -> Thread2 Unlock A
- Thread2 Unlock A -> Thread2 Lock A
- Thread2 Lock A -> Thread1 Unlock A
- Thread1 Unlock A -> Thread1 Unlock B
- Thread1 Unlock B -> Thread1 Lock B

What locking behavior to characterize

- When deadlock happens, threads are attempting to acquire locks while holding another

Thread

Lock A
Lock B
Lock C
….

How do we describe such lock holding-and-attempting's?

Lock dependency – locking order

- Whenever a thread attempts to acquire lock B while hold lock A

Thread

Lock A
Lock B
Unlock B
Unlock A

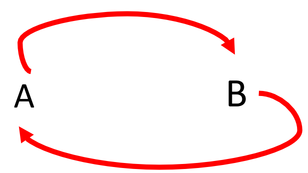
- There exists the locking order

A -> B

Use lock dependency to find deadlocks

The previous example:

Thread1	Thread2
Lock A	Lock B
Lock B	Lock A
↓ characterize	↓ characterize
A -> B	B -> A



Potential deadlock!

Lock dependency vs. waiting relationship

- Whenever a thread attempts to acquire lock B while holding lock A

Thread1	Threadx
Lock A	
Lock B	Lock A
Unlock B	
Unlock A	

- It is very probable there is a Threadx having Lock A, then:

Threadx Lock A -> Thread1 Unlock A

Thread1 Unlock A -> Thread1 Unlock B

Thread1 Unlock B -> Thread1 Lock B

Threadx Lock A -> Thread1 Lock B

A -> B

Lock dependency in general

- Whenever a thread attempts to acquire a lock while hold some locks

Thread1	Thread2	Thread3
Lock A	Lock C	Lock E
Lock B	Lock E	Lock A
Lock C	Lock D	
Lock D	Lock F	
A -> B -> C -> D	C -> E -> D -> F	E -> A

A -> B -> C -> E -> D -> F

A -> B -> C -> E -> D -> F

That is it!

- You have known almost all about what lockdep is doing

- Except
  - What if interrupts come into play?
    - Safe and unsafe lock.
  - Why can it predict?
    - Lock classification
    - Lock waiting pattern
  - Why is there false positive?
    - Incomplete relative timings
    - Exaggerated classification
  - How can it improve performance?
    - Lock chain caching, etc.

Except: when read-write locks come into play

- Read locks (readers) and write locks (writers)
- Exclusive lock vs. read-write lock
  - Write lock == exclusive lock
  - Read locks (only) can be concurrent
- Read-write locks are the general-form locks

But the previous deadlock detection scheme only works for exclusive locks. Why?

Read-write lock design elements

- Can readers be recursive?

```
Thread

RLock A
RLock A
Unlock A
Unlock A
```

- Lock grant order when readers and writers contend

rwsem

- No recursive
- Grant order: prioritize writers

Notation:

Rn: Read lock n
Wn: Write lock n
RRn: Recursive-read lock n
Xn: Read, recursive-read or write lock n

T1	T2	
R1	W2	
W2	R1	No deadlock

T1	T2	T3	
R1	W2		
		W1	
W2	R1		Deadlock

rwlock

- Recursive
- Grant order: prioritize readers

T1	T2	
RR1	W2	
W2	RR1	No deadlock

T1	T2	T3	
RR1	W2		
		W1	
W2	RR1		No deadlock

qrwlock – an implementation of rwlock

- Recursive
- Grant order: readers or writers priority depends on contexts

T1	T2	T3	
(In IRQ)			
RR1	W2		
		W1	
W2	R1		Deadlock

T1	T2	T3	
(In IRQ)			
W1	R2		
		W2	
RR2	W1		No deadlock

The way to general deadlock prediction algorithm

- 7 lemmas lead to (and proves) the solution
- Using abstract cases to show all possibilities

- Step 1: propose the Simple Algorithm
- Step 2: adjust the Simple Algorithm to the Final Algorithm

Lemma1: what makes a deadlock prediction?

- Circular waiting relationship is necessary for deadlocks
- But they are not sufficient with (recursive) read locks

Lemma #1

A deadlock can be and can only be detected at the earliest time at the final arc that completes a waiting circle. In other words, at the final arc, the problem is try to find out whether this circle to come is a deadlock or not.

Lemma2: number of threads in deadlock

- Given a deadlock, assume there are n threads, denoted as:

$T^1, T^2, \ldots, T^n$

- Depending on n, there are two cases:
  - n = 1, referred to as recursion deadlock scenario, skipped
  - n >= 2, grouped as ($T^1, \ldots, T^{n-1}$) and T^n. By consolidating the former, we get a big imagined T1 and T2 (with task numbers adjusted).

Lemma #2

Two threads (T1 and T2) can virtually represent any situation with any number of tasks to check for deadlock.

Locking behavior representation

- Two-thread model (version 1)
  - T1 - the entire previous locking behavior depicted by the lock dependency graph
  - T2 - the current new locking behavior, the X1 -> X2 dependency
- Caveats

Case #1.1

T1	T2	
W1		
RR2	W3	
W3	W1	Deadlock

Case #1.2

T1	T2	
W1		
RR2		
RR2	W3	
W3	W1	No deadlock

Case #1.3

T1a	T1b	T2	
W1	RR2	W3	
RR2	W3	W1	No deadlock

Lemma3: numbers of locks do not matter!

- With the two-thread model, the number of threads does not matter
- Further, the number of locks does not matter either

Lemma #3

Any deadlock scenario can be converted to an ABBA deadlock.

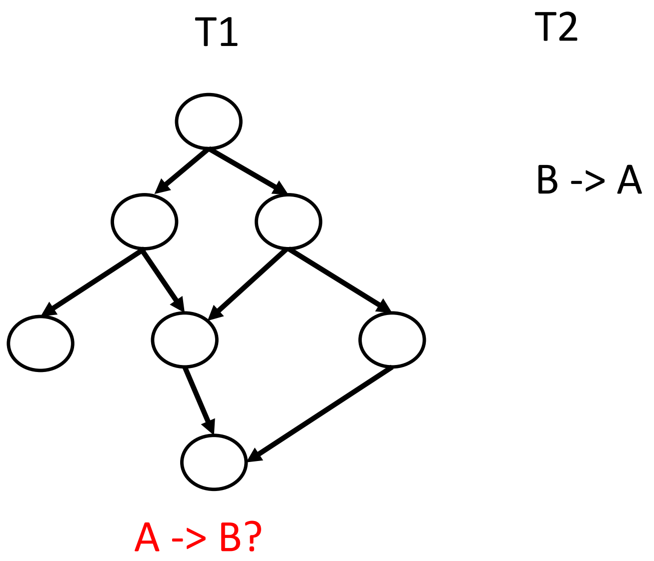
T1	T2
X1	X7
X2	X2
A	B
X3	X3
X4	X8
B	A
X5	
X6	

=

T1	T2
A	B
B	A

General problem to solve

- Given the two-thread model: T1 and T2
- When T2 has a BA to come, is there an AB in T1?



Lock type exclusiveness table

- Lock type
  - Read, recursive-read, or write
- Is A exclusive to B?

T1	T2		T1	T2
		or		
A				A
	B		B	

A vs. B	Recursive-read lock	Read lock	Write lock
Recursive-read lock	No	Yes	Yes
Read lock	No	Yes	Yes
Write lock	Yes	Yes	Yes

Simple Algorithm

- Given T2's dependency, divide and conquer the cases respectively
- There are 9 cases
  - Read lock and read lock
  - Read lock and recursive-read lock
  - Read lock and write lock
  - Write lock and read lock
  - Write lock and recursive-read lock
  - Write lock and write lock
  - Recursive-read lock and read lock
  - Recursive-read lock and recursive-read lock
  - Recursive-read lock and write lock

Simple Algorithm case

- When the final dependency is ended with read lock and read lock

Case #2.1

T1	T2	
X1	R2	
W2	R1	Deadlock

Case #2.2

T1	T2	
X1	R2	
R2	R1	Deadlock

Case #2.3

T1	T2	
X1	R2	
RR2	R1	No deadlock

Simple Algorithm cont.

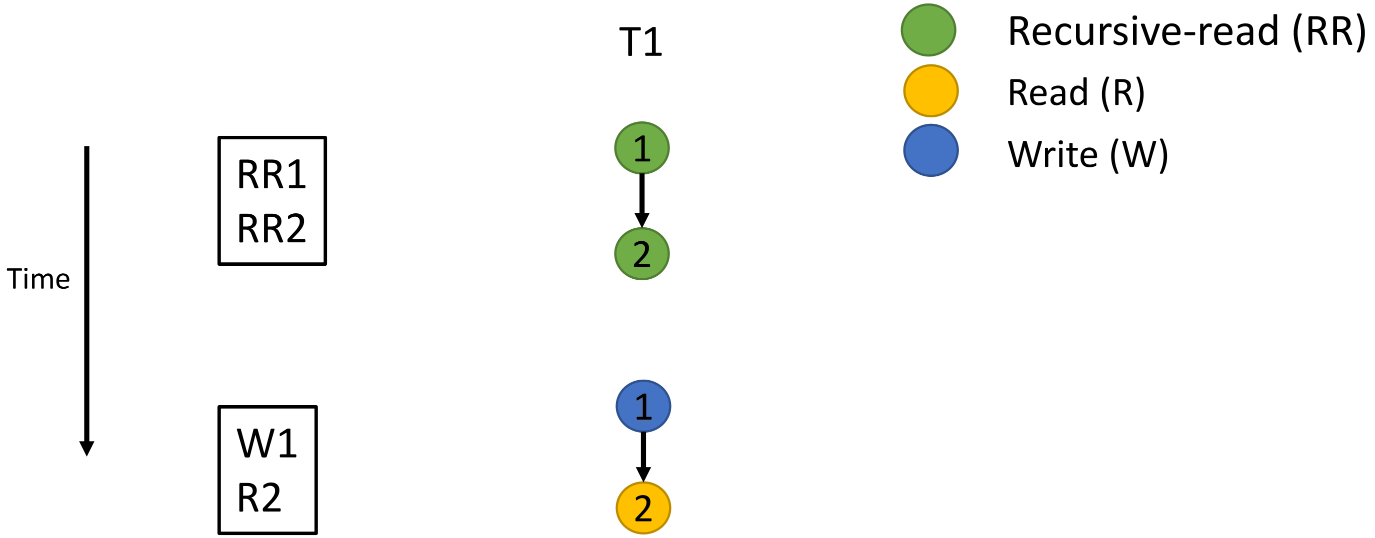
- Given T1 and T2, and an ABBA locks

T1	T2
X1.A	X2.A
X2.B	X1.B

- Simple algorithm
  - (1) If X1.A vs. X1.B are exclusive and X2.A vs. X2.B are exclusive then it is deadlock
  - (2) Otherwise no deadlock

Lock type promotion

- In locking Lock type in dependency is promoted toward more exclusiveness



Final Algorithm (1/6)

- A case fails the Simple Algorithm

Case #13			**Case #12**		
T1	T2		T1	T2	
X1			X1	X3	
RR2	RR2		RR2	RR2	
X3	X1	No deadlock	X3	X1	Deadlock

Indirect dependency: X3 -> X1

Final Algorithm (2/6)

- Direct dependency vs. indirect dependency

A -> B -> C ➡ Direct dependency:
(1) A -> B
(2) B -> C
Indirect dependency:
(1) A -> C

Lemma #4

The direct dependency (e.g., RR2 -> X1 in Case #12) that serves in the path from X3 -> X1 is **critical.**

Final Algorithm (3/6)

- Case #12 vs. Case #13

> **Lemma #5**
>
> Missed in Case #13, the game changer to Case #12 is that it has X3 in T2 whereas Case #13 does not.

- From the Simple Algorithm to the Final Algorithm
  - (a) Continue searching the graph to find a new circle.
  - (b) In the new circle, do the Simple Algorithm, if a deadlock is found, done
  - (c) In the new circle, if previous locks in T2's stack (or chain) are in it and then check whether the circle is indeed a deadlock. This is done by checking each newly introduced indirect dependency (such as X3 -> X1 in Case #12) according to our Simple Algorithm as before.
  - (d) If a deadlock is found then the algorithm is done, otherwise go to (a) unless the entire graph is traversed

Final Algorithm (4/6)

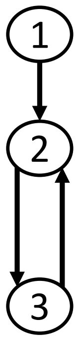
- The failed case can be solved now

Case #12

T1	T2
X1	X3
RR2	RR2
X3	X1

T1

Previous dependency graph



T2

New dependency 2 -> 1

First: Do we have 1 -> 2 ? Yes
Second: do we have 1 -> 3? Yes.

Final Algorithm (5/6)

- Why does the Final Algorithm work?

> **Lemma #6**
>
> Lemma #5 nicely raises a question whether a previous lock involved indirect dependency in T2 is **necessary**. The answer is yes, otherwise our **Simple Algorithm** has already solved the problem.

- Modified two-thread model (version 2)
  - T1 - the entire previous locking behavior depicted by the lock dependency graph.
  - T2 - the current task's held lock stack worth of direct and indirect dependencies.

Final algorithm (6/6)

- Why does the final algorithm work?

> **Lemma #7**
>
> It is also natural to ask whether indirect dependencies with a starting lock in T2 only is **sufficient**: what if the indirect dependency has a starting lock from T1. The answer is yes too.

- More explanation
  - Because Lemma #2 and Lemma #3 say that any deadlock is an ABBA so that T1 can only contribute an AB and T2 must have a BA, and the final direct arc or dependency is the BA or the ending part of the BA.

Questions?

Thanks,
Yuyang Du

Backup

- Backup