Recursive read deadlocks and Where to find them

冯博群 Boqun Feng
boqun.feng@gmail.com
Agenda

- Deadlock cases
- Lockdep
- Flavors of read/write locks
- More deadlock cases
- (Recursive) read deadlock detection
Deadlock cases

- self deadlock

```c
P0
spin_lock(&A);
...
spin_lock(&A);
```

- ABBA deadlock

```c
P0
spin_lock(&A);
...
spin_lock(&B);
spin_lock(&B);
spin_lock(&A);
P1
spin_lock(&B);
...
spin_lock(&A);
```
Deadlock cases (cont.)

- IRQ safe->unsafe deadlocks
  - IRQs bring more "code combinations"

```c
P0
<irq_enabled>
spin_lock(&A);
...
<in irq handler>
spin_lock(&A);

P0
<irq_enabled>
spin_lock(&A);
...
<in irq handler>
spin_lock(&A);

P1
<irq_disabled>
spin_lock(&B);
...

spin_lock(&A);
```
### Deadlock cases (cont.)

- **ABBCCA deadlocks**
  - or more

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>spin_lock(&amp;A);</td>
<td>spin_lock(&amp;B);</td>
<td>spin_lock(&amp;C);</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>spin_lock(&amp;B);</td>
<td>spin_lock(&amp;C);</td>
<td>spin_lock(&amp;A);</td>
</tr>
</tbody>
</table>
Lockdep

- Locks are grouped by classes
- Lock dependency
  - A -> B

- Dependency graph

```
spin_lock(&A);
...
spin_lock(&B);
```
Lockdep (cont.)

- Deadlock detection
  - A closed path (circle) in the dependency graph
Flavors of read/write locks

- Recursive/unfair rwlocks
  - readers are preferable
Flavors of read/write locks (cont.)

- Non-recursive/fair rwlocks
## Flavors of read/write locks (cont.)

<table>
<thead>
<tr>
<th>flavors</th>
<th>multiple readers</th>
<th>recursive c.s</th>
<th>a reader blocks another reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>non-recursive</td>
<td>Y</td>
<td>N</td>
<td>Y* (via a waiting writer)</td>
</tr>
</tbody>
</table>
Flavors of read/write locks (cont.)

- **Block condition**
  - Recursive readers can get blocked by writers
  - Non-recursive readers can get blocked by non-recursive readers (via a waiting writer) or writers

<table>
<thead>
<tr>
<th></th>
<th>reader(recursive or not)</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive reader</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>non-recursive(r &amp; w)</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
More deadlock cases

- For non-recursive read/write locks
  - Same as spinlocks, since readers can block each other via a waiting writer

```
// P0
read_lock(&A);
...  
spin_lock(&B);

// P1
spin_lock(&B);
...  
read_lock(&A);

// P2
write_lock(&A);
```
More deadlock cases

- For recursive locks, things get interesting:
  - This is not a deadlock

```
read_lock(&A);       spin_lock(&B);
...                  ...
spin_lock(&B);       read_lock(&A);
```
More deadlock cases

- But this is a deadlock

```
P0
  read_lock(&A);
  ...
  spin_lock(&B);

P1
  spin_lock(&B);
  write_lock(&A);
```
More deadlock cases

- Things get complicated when we mixed recursive and non-recursive read locks
- queued rwlock
  - non-recursive read lock in process context
  - recursive read lock in irq context
More deadlock cases

- Recursive deadlock case

```
P0                   P1                  P2

<in irq handler>
read_lock(&B);      spin_lock_irq(&A);
spin_lock(&A);      read_lock(&B);
write_lock_irq(&B);
```
More deadlock cases

- Recursive *not* deadlock case

```c
<in irq handler>
spin_lock(&A);
read_lock(&B);
read_lock(&B);
spin_lock_irq(&A);
```

```
write_lock_irq(&B);
```
Recursive read deadlock detection

- Limitation of current lockdep
  - circles mean deadlocks
  - while not all the circles mean deadlocks if we consider recursive readers.
Recursive read deadlock detection

● Goals
  ○ Compatible with original lockdep detection.
  ○ Handle qrwlock semantics.
  ○ No false positive.
Recursive read deadlock detection

● Overview
  ○ Classification for lock dependencies
  ○ Definition of "strong" dependencies
  ○ Deadlock Condition
  ○ Informal Proof
Classification of lock dependencies

- We used to treat all lock dependencies as the same but they are really not.
- \{R \text{reader, reader, writer}\} \rightarrow \{R \text{reader, reader, writer}\} : 9 combinations
Classification of lock dependencies

- Groups things into 4
  - \{R reader, reader\} -> \{reader, writer\}: -(SN)->
  - \{R reader, reader\} -> \{R reader\}: -(SR)->
  - \{writer\} -> \{reader, writer\}: -(EN)->
  - \{writer\} -> \{R reader\}: -(ER)->
- Why? Because for a dependency A -> B, we cares:
  - Whether A can block anyone
  - Whether B can get blocked by anyone

```
read_lock(&A);
...
spin_lock(&B);
write_lock(&A);
```

Diagram:

```
A  --S--  B
  |     |
  |     |
  |     |
  W     EN
```
Definition of "strong" dependencies

- Chaining lock dependencies via block conditions
- For dependencies A -> B and B -> C
  - A -> B -> C is a "strong" dependency path iff
    - A -> B : -(*R)-> and B -> C : -(E*)->, or
    - A -> B : -(*N)-> and B -> C : -(S*)->, or
    - A -> B : -(*N)-> and B -> C : -(E*)->
  - IOW, -(*R)-> -(S*)-> will break the dependency
- works for "A -> B, B -> C and C -> D" case, and so on
Deadlock condition

- A strong dependency chain/path forms a circle

```
P0                   P1                  P2

spin_lock(&A);       read_lock(&B);      read_lock(&C);
...                  ...                 ...
write_lock(&B);      write_lock(&C);     spin_lock(&A);
```
Informal Proof

● We want to prove:
  ○ A strong dependency circle is equivalent to deadlock possibility

● Necessary condition
  ○ a strong dependency circle => deadlock possibility
  ○ Easy, because a strong dependency circle means we can build a combination of locking sequences that cause deadlock.
Informal Proof (cont.)

- **Sufficient condition**
  - deadlock possibility => a strong dependency circle
  - My trick
    - deadlock possibility => circular wait (deadlock necessary condition according to wikipedia)
    - circular wait => a strong dependency circle
Implementation

- Extend __bfs() to walk on strong dependency path
- Make LOCK*STATE* part of the chainkeys
- Add test cases
  - also unleash irq_read_recursion2
- Enable this for srcu

Code
- git.kernel.org/pub/scm/linux/kernel/git/boqun/linux.git arr-rfc-wip