Recursive read deadlocks and Where to find them

冯博群 Boqun Feng
Agenda

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

- self deadlock

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

- ABBA deadlock

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

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

<table>
<thead>
<tr>
<th>P0</th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;irq enabled&gt;</td>
<td>&lt;irq enabled&gt;</td>
<td>&lt;irq disabled&gt;</td>
</tr>
<tr>
<td>spin_lock(&amp;A);</td>
<td>spin_lock(&amp;A);</td>
<td>spin_lock(&amp;B);</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>&lt;in irq handler&gt;</td>
<td>&lt;in irq handler&gt;</td>
<td></td>
</tr>
<tr>
<td>spin_lock(&amp;A);</td>
<td>spin_lock(&amp;B);</td>
<td>spin_lock(&amp;A);</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Deadlock cases (cont.)

- ABBCCA deadlocks
  - or more

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

- Detect deadlock possibility
  - Assume the worst case: all the combinations of code sequences can happen

- Lock dependency
  - A -> B
  - Assume we can catch all dependencies

- 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>reader(recursive or not)</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive reader</td>
<td>No</td>
</tr>
<tr>
<td>non-recursive(r &amp; w)</td>
<td>Yes</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               P1                     P2
read_lock(&A);   spin_lock(&B);      write_lock(&A);
...              ...                 ...
spin_lock(&B);   read_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

```c
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**

```c
<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);
write_lock_irq(&B);
read_lock(&B);      spin_lock_irq(&A);
```
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
Classification of lock dependencies

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

- (S)hared locks: reader (recursive or not)
- (E)clusive locks: writer (or plain spinlocks)
- (R)ecursive readers
- (N)on-recursive (readers and writers)

<table>
<thead>
<tr>
<th>blocked by</th>
<th>reader(recursive or not)</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursive reader</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>non-recursive(r &amp; w)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>blocked by</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>N</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Classification of lock dependencies

- Groups things into 4
  - -(SN)-> : \{R reader, reader\} -> \{reader, writer\}
  - -(SR)-> : \{R reader, reader\} -> \{R reader\}
  - -(EN)-> : \{writer\} -> \{reader, writer\}
  - -(ER)-> : \{writer\} -> \{R reader\}

- Why? Because for a dependency A -> B, we cares:
  - Whether A can block anyone
  - Whether B can get blocked by anyone

<table>
<thead>
<tr>
<th>blocked by</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>N</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

P0

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

P1

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

<table>
<thead>
<tr>
<th>blocked by</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>N</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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);
```
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