Protecting Locks Against Unbalanced Unlock()
Locks

- Provide mutual exclusion for shared data
- Most popular mutual-exclusion primitive
- Common usage:

```java
m.Lock()
```

Critical Section (CS)

```java
m.Unlock()
```
Many Locking Algorithms

- Tens of lock algorithms over the past couple of decades

All focus on performance
Our Focus: Lock Misuse

```go
if cond {
    m.Lock()
}

m.Unlock()
```
Problem: Unbalanced Unlock

• Accidental call to `unlock()` without `lock()`

• Impact
  • Mutex violation?
  • Starvation?
  • Corruption of lock internals?
  • Program corruption?
  • Benign?

• Can we
  • detect unbalanced-unlock?
  • devise/alter lock algorithms to avoid problematic situations?

Analysis of and remedy to popular spinlocks
Contributions

- Show unbalanced-unlock is a common problem
- Analyze popular locks in unbalanced-unlock situations
- Remedy popular locks to be resilient to unbalanced-unlock
- Show remedied lock designs remain performant
Unbalanced-unlock in the Linux Kernel

```c
if (wilc->quit){
    goto out;
}
mutex_lock(...);
tqe = ...;
if (!tqe){
    goto out;
}
...
out:
mutex_unlock(...);
return ret;
```

Linux Kernel code: drivers/staging/wilc1000/wlan.c (commit id: bd4217c)
Unbalanced-unlock in the Open-Source

Categorization of Misuse Type in Lock Related Code Changes

- Golang: 20 (14 Unbalanced-unlock, 6 Unbalanced-lock)
- Linux kernel: 40 (12 Unbalanced-unlock, 28 Unbalanced-lock)
- LLVM project @ Intel: 26 (16 Unbalanced-unlock, 10 Unbalanced-lock)
- MySQL: 7 (4 Unbalanced-unlock, 3 Unbalanced-lock)
- memcached: 9 (3 Unbalanced-unlock, 6 Unbalanced-lock)
Unbalanced-lock: forgetting to call unlock

```go
if cond {
    m.Lock()
    ...
    return;
}

m.unlock()
```

Well-known problem
How do different locks fare in the presence of unbalanced-unlock?

**Notation:** Tm denotes thread that misbehaves and Tx denotes all other threads

<table>
<thead>
<tr>
<th>Lock</th>
<th>Violates Mutex</th>
<th>Starves Tm</th>
<th>Starves Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAS</td>
<td>✓</td>
<td>X</td>
<td>NA</td>
</tr>
<tr>
<td>Ticket</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Anderson ABQL</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Graunke-Thakker</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>MCS</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>CLH</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>MCS-K42</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hemlock</td>
<td>X</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>HMCS</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>HCLH</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C-RW-NP/RP/WP</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Peterson’s lock</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Fisher’s lock</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Lamport’s lock</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>
Test and Set (TAS) Lock

lock object L, shared-variable / Global

lock L: UNLOCKED

T1: lock()  Tx: lock()

spin

T1: unlock()  Tm: unlock()

Unbalanced-Unlock

T1 and Tx are both in CS. Violation of mutual exclusion!
Test and Set Lock Analysis

<table>
<thead>
<tr>
<th>Lock</th>
<th>Violates Mutex</th>
<th>Starves Tm</th>
<th>Starves Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAS</td>
<td>✓</td>
<td>✗</td>
<td>NA</td>
</tr>
</tbody>
</table>

- Mutual exclusion is violated
  - every instance of unbalanced-unlock releases *at most one* waiting thread into CS

- No starvation
  - thread involved in unbalanced-unlock (Tm) returns from the call to `unlock()`
  - By design, TAS lock does not ensure starvation freedom
Test and Set Lock - Remedy

- Intuition: store the PID (unique thread identifier) of the current lock holder instead of the flag (LOCKED/UNLOCKED) in the lock

```c
lock L: ULONG_MAX
```

```c
t1: lock()
```

```c
t1: unlock()
```

```c
tm: unlock()
```

Caller PID is m, stored PID (in L) is 1. There is a mismatch.

```c
unlock(unsigned long tid) {
    if L is tid
        set L to ULONG_MAX; return true
    return false
}
```

Unbalanced - Unlock

```c
tid = m
```

```c
tm: unlock()
```

Unbalanced-Unlock
MCS Lock: Analysis and Remedy
MCS Lock - Analysis

lock L: **NULL**

T1: lock(t1)

T2: lock(t2)

T3: lock(t3)

lock L: **NULL**

T1: unlock(t1)

T2: unlock(t2)

T3: unlock(t3)

- **Caution:** before resetting,
  - Check if L still points to t3 (no successor has appeared in the meanwhile). If not:
    - wait till the successor appears in t3->next
    - set the successor’s locked to false and return

swaps the lock with itself, gets the predecessor, and attaches itself to predecessor

- No successors / waiters for the lock. Reset L to NULL.
MCS Lock - Analysis

lock L: NULL

node objects still exist and the fields are not reset. Links may exist.
MCS Lock - Analysis (Scenario 1)

- **Earlier** -  
  \[\text{lock } L: \text{ NULL}\]

  - t1: Locked  
    - Next
  - t2: \text{false}  
    - Next
  - t3: \text{false}  
    - Next

- **Now** - suppose T3 is holding the lock and T2 is spinning:

  \[\text{lock } L: \text{ false} \]

  - t3: \text{false}  
    - Next
  - t2: \text{false}  
    - Next

- **Next** - T1: unlock(t1) (unbalanced-unlock!)

  - T1 sets t1->next->locked to false.

  T3 and T2 are both in CS. **Violation of mutual exclusion.**
MCS Lock - Analysis (Scenario 2)

- **Earlier**
  - `lock L`: NULL
  - t1: Locked Next
  - t2: false Locked Next
  - t3: false Locked Next

- **Now** - T3: `unlock(t3)` *(unbalanced-unlock!)*
  - No successors / waiters for the lock. Reset L to NULL.
    - before resetting,
      - Check if L still points to t3 (no successor has appeared in the meanwhile). If not:
        - wait till the successor appears in t3->next

This is never going to happen! T3 starves.
MCS Lock - Remedy

- Intuition: maintain an invariant that a flag (Locked) should be true whenever the releaser wants to release the lock.

Initialize, reset and check Locked

```
lock L:
T1: lock(t1)
t1: false
Locked Next
T2: lock(t2)
t2: true NULL
Locked Next
T1: unlock(t1)
Tm: unlock(tm) {
  if (locked == false)
    return false
...
```
CLH Lock: Analysis and Remedy
CLH Lock - Analysis

lock L: 

T1: lock(t1)  T2: lock(t2)

T1: unlock(t1)  T2: unlock(t2)

prev  must_wait  prev  must_wait  prev  must_wait
CLH Lock - Analysis (Scenario 1)

- *Earlier* - 
  - lock L: 
    - t1: false
    - t2: false
    - prev must_wait: false
    - bootstrapp: false

- *Now: suppose T1 is holding the lock and T3 is spinning:*
  - lock L: 
    - bootstrap: false
    - t2: false
    - t1: false
    - t3: true
    - prev must_wait: false
    - prev must_wait: false
    - prev must_wait: false

- *Next: T2: unlock(t2) (unbalanced-unlock!)*
  - T2 sets t2->must_wait = false
  - Takes ownership of t1! (predecessor of t2)

  T2: unlock(t2) now releases T3 from spinning: T3 and T1 are both in CS. Violation of mutual exclusion.
CLH Lock - Analysis (Scenario 2)

- Extension of scenario 1 from previous slide

lock L:

bootstrap: false \rightarrow t2: false \rightarrow t1: false \rightarrow t3: true

prev must_wait \rightarrow prev must_wait \rightarrow prev must_wait

- T2: unlock(t2) and T1: unlock(t1) racily update the must_wait field
- The updates may be lost preventing waiting threads from getting the lock. **Successors starve!**
CLH Lock - Remedy

- Intuition: maintain an invariant that prev pointer is not null only when a lock is being held
  - Initialize, reset and check prev

- After an episode of successful lock-unlock:
  ```
  lock L:
  t1:
  prev NULL false
  must_wait
  t2:
  prev NULL false
  must_wait
  bootstrap:
  prev NULL false
  Tm: unlock(tm) {
  ...
  tm->prev = NULL
  return true
  ...}
  Detects and prevents unbalanced-unlock
  ```
Fischer’s Software Lock

```
start:
  while <x != 0>;
  <x := i>;
  <delay>
  if <x != i> goto start;

  critical section;

  if <x != i> goto exit;

  x := 0

exit:
```

lock()
More Locks, Analysis and Remedies...

- Hierarchical locks
- Reader-Writer locks
- Reentrant Locks
- Hemlock
- MCS-K42 lock
- Software locks
Experimental Setup

• Configuration
  • dual-socket system
  • 24-core, Intel Xeon Gold 6240C@2.60GHz processor
  • CPU has 64 KB shared data and instruction caches
  • 1 MB unified L2 and 36 MB L3 unified caches
  • 384GB DDR4 memory
  • Rocky Linux 9

• Benchmarks
  • SPLASH-2x [6] and PARSEC 3.0 [5]
  • barnes, dedup, ferret, fluidanimate, fmm, ocean, radiosity, raytrace, and streamcluster
  • Native input dataset
## Results

**Takeaway:** Overhead of proposed remedy for lock algorithms is negligible (<5%)

<table>
<thead>
<tr>
<th></th>
<th>-0.14</th>
<th>1.04</th>
<th>-0.12</th>
<th>0.54</th>
<th>0.93</th>
<th>1.18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes (48)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dedup (48)</td>
<td>-1.59</td>
<td>3.47</td>
<td>-3.32</td>
<td>0.32</td>
<td>4.25</td>
<td>1.62</td>
</tr>
<tr>
<td>Ferret (48)</td>
<td>-0.31</td>
<td>0.42</td>
<td>-0.45</td>
<td>-0.05</td>
<td>1.45</td>
<td>-0.97</td>
</tr>
<tr>
<td>Fluidanimate (32)</td>
<td>0.19</td>
<td>2.8</td>
<td>-0.78</td>
<td>1.96</td>
<td>NA</td>
<td>1.96</td>
</tr>
<tr>
<td>FMM (48)</td>
<td>0</td>
<td>0.64</td>
<td>-0.29</td>
<td>-0.85</td>
<td>0.4</td>
<td>-0.29</td>
</tr>
<tr>
<td>Ocean (32)</td>
<td>1.68</td>
<td>4.23</td>
<td>3.79</td>
<td>0.94</td>
<td>3.31</td>
<td>0.55</td>
</tr>
<tr>
<td>Radiosity (48)</td>
<td>2.08</td>
<td>19.5</td>
<td>0.87</td>
<td>2.62</td>
<td>1.72</td>
<td>-0.88</td>
</tr>
<tr>
<td>Raytrace (48)</td>
<td>16.9</td>
<td>86.7</td>
<td>3.08</td>
<td>-0.89</td>
<td>2.83</td>
<td>2.38</td>
</tr>
<tr>
<td>Streamcluster (48)</td>
<td>1.3</td>
<td>61.3</td>
<td>1.72</td>
<td>1.13</td>
<td>NA</td>
<td>-2.17</td>
</tr>
<tr>
<td>Synthetic (48)</td>
<td>22</td>
<td>118</td>
<td>-0.15</td>
<td>3.2</td>
<td>3.27</td>
<td>1.64</td>
</tr>
</tbody>
</table>

Numbers indicate overhead percentage at maximum thread count (48)
Conclusions

• **Unbalanced-unlock** is surprisingly common in popular open-source repositories.

• A systematic analysis of popular locks in unbalanced-unlock situation shows:
  • Mutex violation
  • Starvation
  • Corruption of lock internals and program
  • sometimes be side-effect free

• Remedy to eliminate side effects are simple and we apply the remedy to a representative set of lock implementations

• The modified lock implementations did not significantly affect performance


