Protecting Locks Against Unbalanced Unlock()

Vivek Shahare [1] Milind Chabbi [2,1] Nikhil Hegde [1]

[1] Indian Institute of Technology Dharwad, India

[2] Programming Systems Group, Uber Technologies Inc., USA

॥ सा विद्या या विमुक्तये ॥

भारतीय प्रौद्योगिकी संस्थान धारवाड

Indian Institute of Technology Dharwad

Locks

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

```
m.Lock()
```

Critical Section (CS)

```
m.Unlock()
```

Many Locking Algorithms

- Tens of lock algorithms over the past couple of decades

Test and Set (TAS)
TAS-BO
Ticket
Anderson's Array-based
Reader-Writer
Graunke and Thakker
MCS
TATAS
HBO
CLH
MCS-K42
Cohort
HCLH
HMCS
Hemlock
BRAVO
CNA
Flat-combining
Fissile
Delegation
Malthusian
FLOCK
TWA

All focus on performance

Our Focus: Lock Misuse

```
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

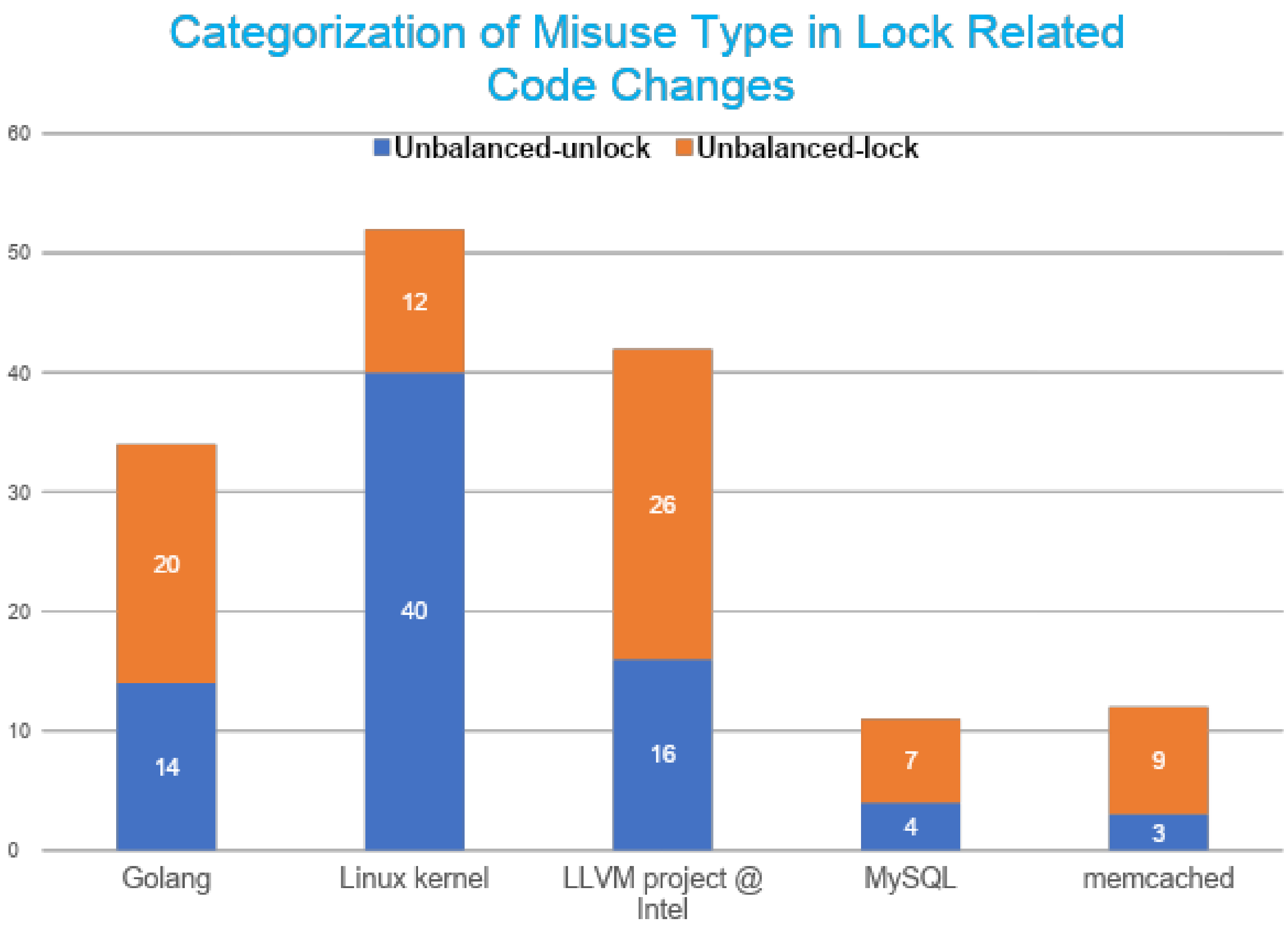
Unbalanced-unlock in the Linux Kernel

```
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



Unbalanced-lock: forgetting to call unlock

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

Well-known problem

Lock Protocol Analysis - Summary

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

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

Lock	Violates Mutex	Starves Tm	Starves Others
TAS	✓	✗	NA
Ticket	✓	✗	✓
Anderson ABQL	✓	✗	✗
Graunke-Thakker	✗	✗	✓
MCS	✓	✓	✗
CLH	✓	✗	✓
MCS-K42	✓	✓	✓
Hemlock	✗	✓	✗
HMCS	✓	✓	✗
HCLH	✗	✗	✗
C-RW-NP/RP/WP	✓	✗	✓
Peterson's lock	✗	✗	✗
Fisher's lock	✓	✗	✗
Lamport's lock	✓	✗	✓

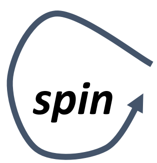
Test and Set (TAS) Lock

lock object L, shared-variable / Global

lock L: UNLOCKED

T1: lock()

Tx: lock()



T1 and Tx are both in CS.
Violation of mutual exclusion!

T1: unlock()

Tm: unlock()

Unbalanced-Unlock

Test and Set Lock Analysis

Lock	Violates Mutex	Starves Tm	Starves Others
TAS	✓	✗	NA

- 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

lock L: ULONG_MAX

T1: lock()

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

T1: unlock()

tid = m

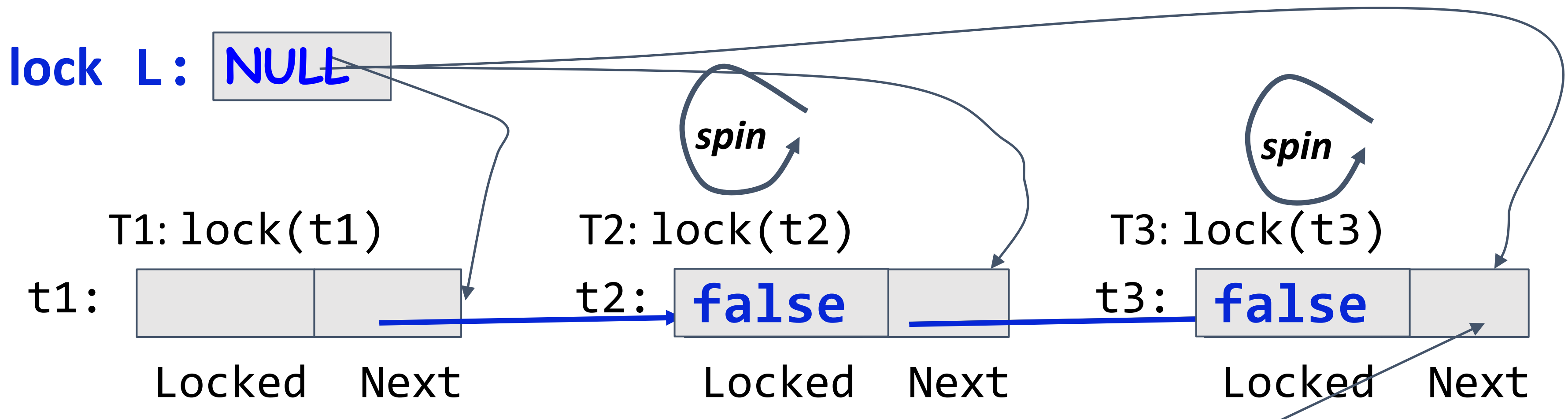
Tm: unlock()

Unbalanced-Unlock

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

MCS Lock: Analysis and Remedy

MCS Lock - Analysis



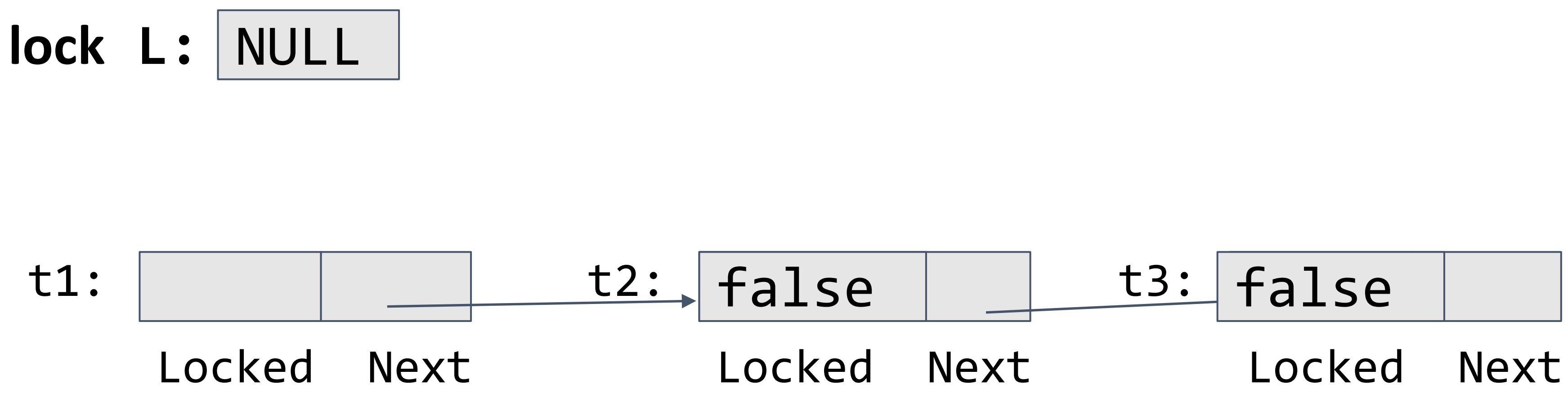
T1: unlock(t1) T2: unlock(t2) T3: unlock(t3)

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

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

  - **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

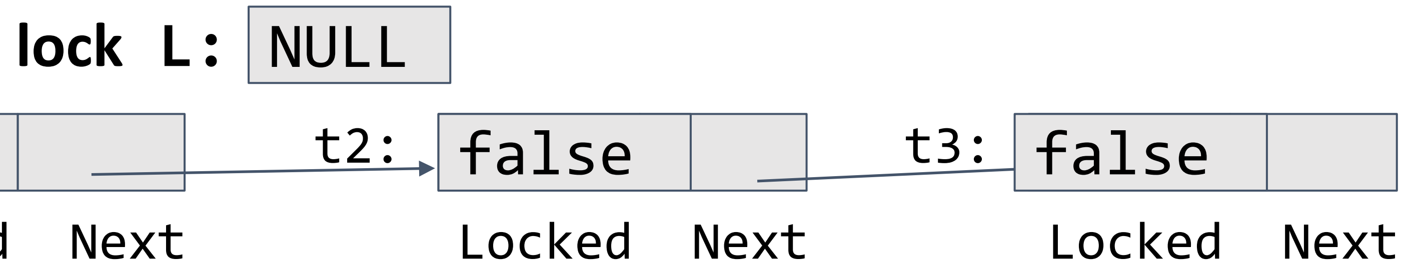
MCS Lock - Analysis



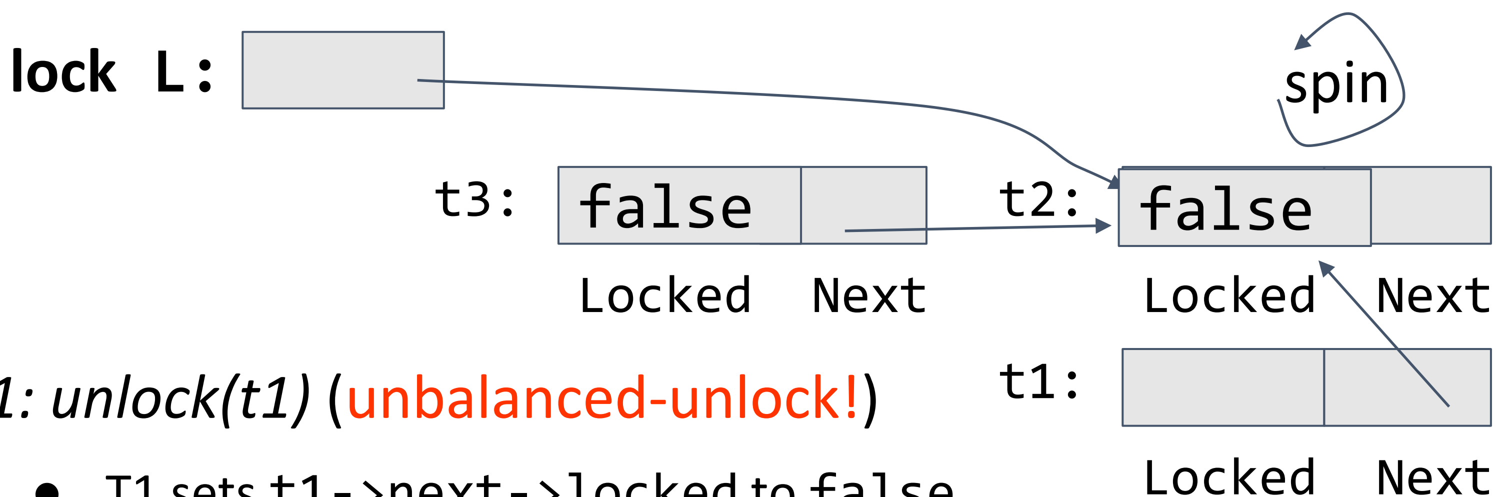
node objects still exist and the fields are not reset. Links may exist.

MCS Lock - Analysis (Scenario 1)

- *Earlier -*



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

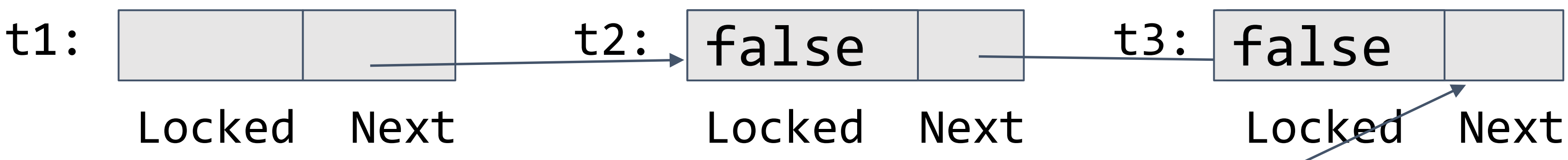


- *Next: T1: unlock(t1)* (unbalanced-unlock!)
  - T1 sets `t1->next->locked` to `false`.

T3 and T2 are both in CS. Violation of mutual exclusion.

MCS Lock - Analysis (Scenario 2)

- *Earlier -* **lock L:** NULL



- *Now - T3: unlock(t3)* (unbalanced-unlock!)
  - No successors / waiters for the lock. Reset L to NULL.
    - before resetting,

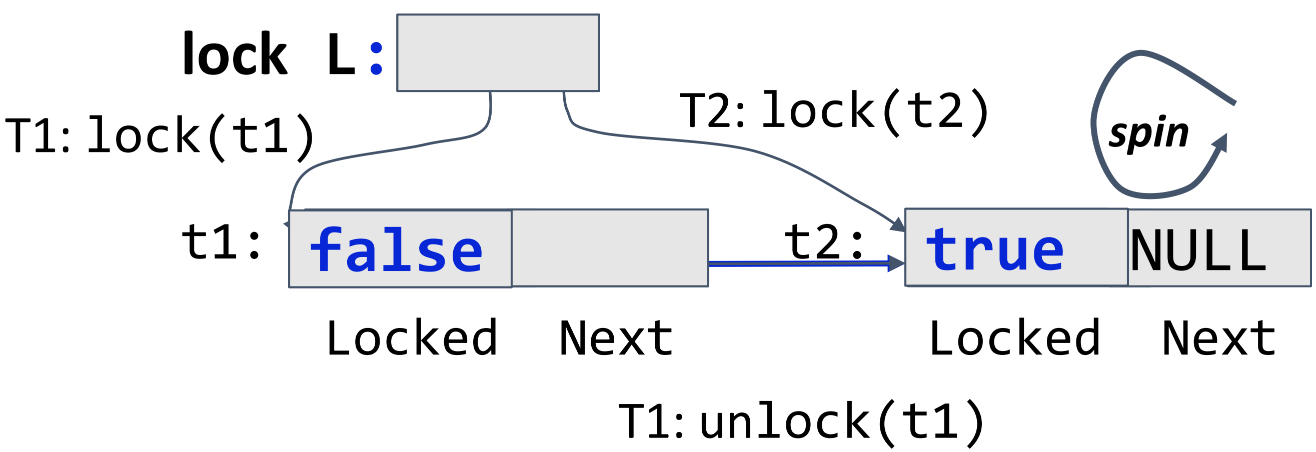
      Check if L still points to t3 (no successor has appeared in the meanwhile). If not:
      - wait till the successor appears in t3->next

This is never going to happen! T3 starves.

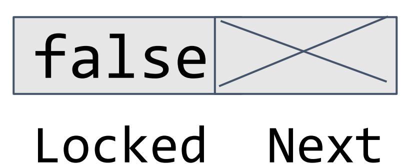
MCS Lock - Remedy

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

Initialize, reset and check `Locked`

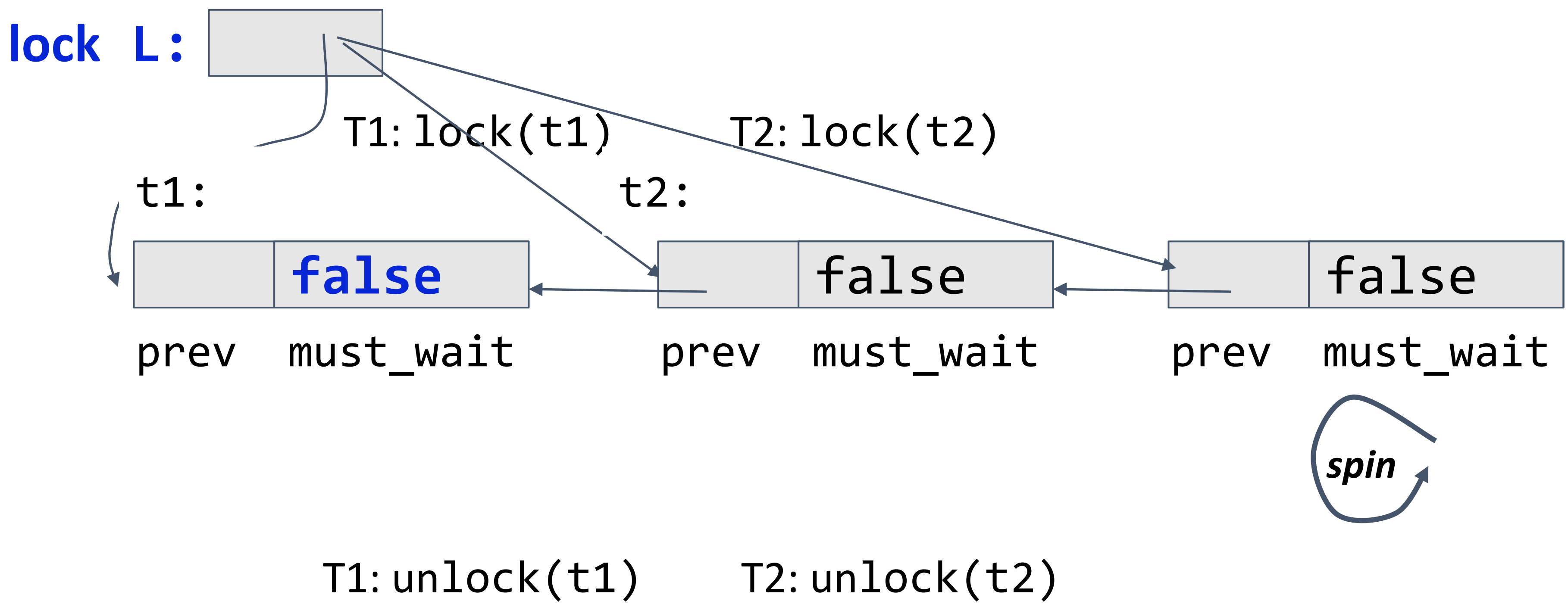



```
Tm: unlock(tm) {
    if (locked == false)
            return false
...
```



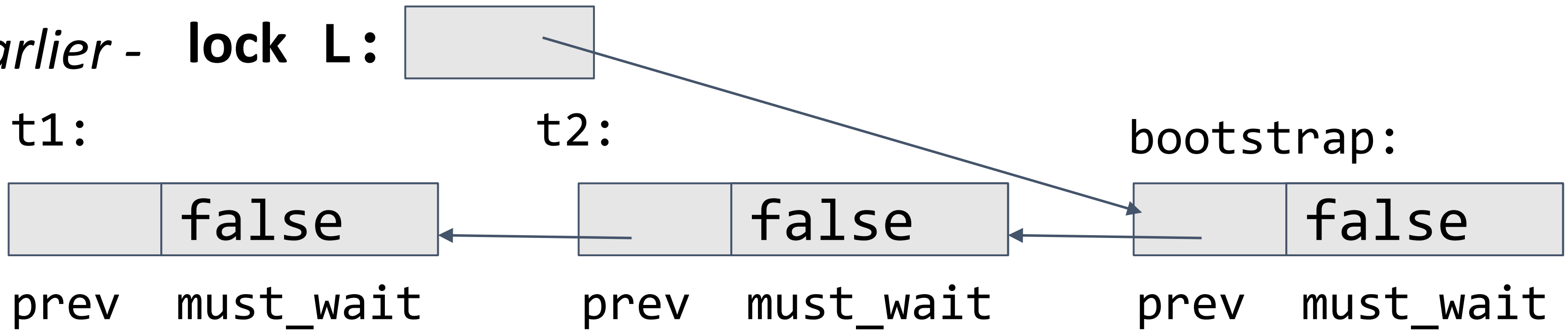
CLH Lock: Analysis and Remedy

CLH Lock - Analysis

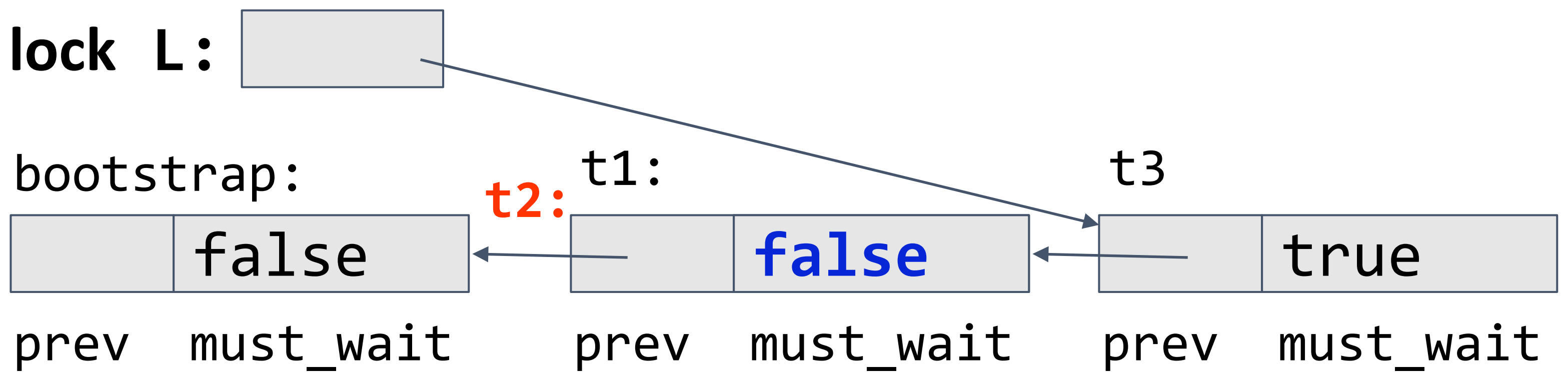


CLH Lock - Analysis (Scenario 1)

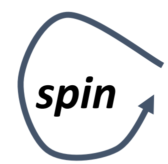
- *Earlier -*



- *Now: suppose T1 is holding the lock and T3 is spinning:*



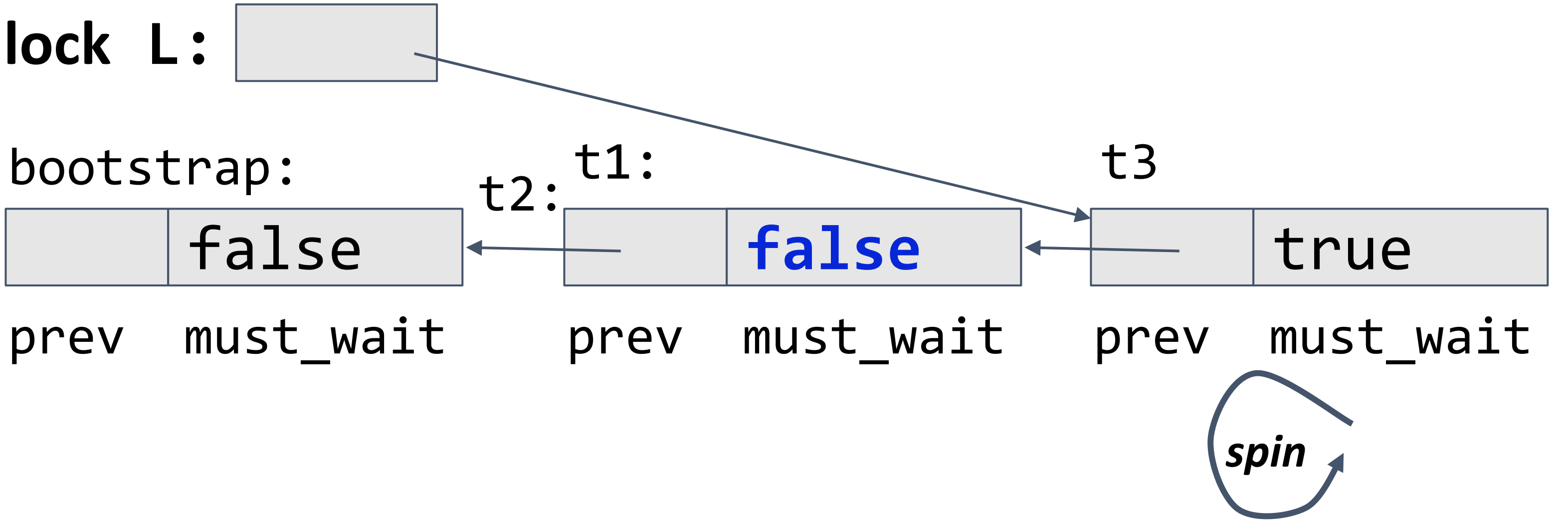
- *Next: T2: unlock(t2)* (unbalanced-unlock!)
  - T2 sets `t2->must_wait = false`
  - Takes ownership of `t1`! (predecessor of `t2`)



T2: unlock(t2) now releases T3 from spinning : T3 and T1 are both in CS. Violation of mutual exclusion.

CLH Lock - Analysis (Scenario 2)

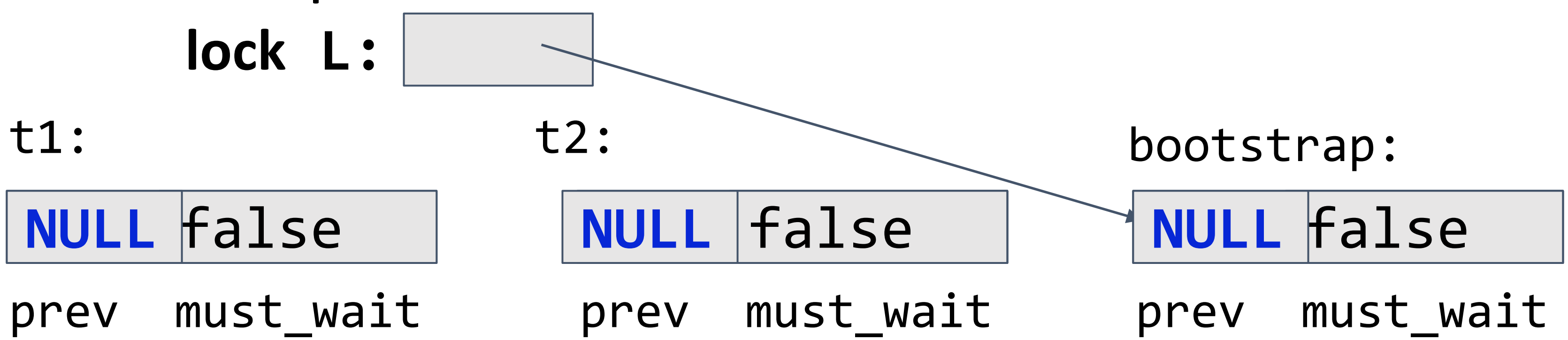
- *Extension of scenario 1 from previous slide*



- 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`:




```
Tm: unlock(tm) {
    ...
      tm->prev = NULL
      return true
```

```
Tm: unlock(tm) {
      if(tm->prev == NULL)
        return false
    ...
```

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()

unlock()

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]
  - $barnes$, $dedup$, $ferret$,$fluidanimite$, fmm, $ocean$, $radiosity$, $raytrace$, and $streamcluster$
  - $Native$ input dataset

Results

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

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

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

References

1. Spinlocks. (n.d.). www.cs.rochester.edu. Retrieved April 14, 2022, from https://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html
2. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9(1):21–65, 1991
3. Queue locks on cache coherent multiprocessors. International Parallel Processing Symposium, pages 26-29, 1994.
4. Hugo Guiroux. 2018. LiTL: Library for Transparent Lock interposition. https://github.com/multicore-locks/litl
5. Dave Dice and Alex Kogan. 2021. Hemlock: Compact and Scalable Mutual Exclusion. In Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21). New York, NY, USA, 173–183.
6. Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. ACM SIGPLAN Notices 50, 8 (2015),215–226. Anders Landin and Eric Hagersten.
7. Christian Bienia. 2011. Benchmarking Modern Multiprocessors. Ph.D. Dissertation. Princeton University.
8. PARSEC Group et al. 2011. A memo on exploration of SPLASH-2 input sets. Princeton University (2011).
9. Synchronization Constructs - OMSCS Notes; www.omscs-notes.com. Retrieved April 12, 2022, from https://www.omscs-notes.com/operating-systems/synchronization-constructs/