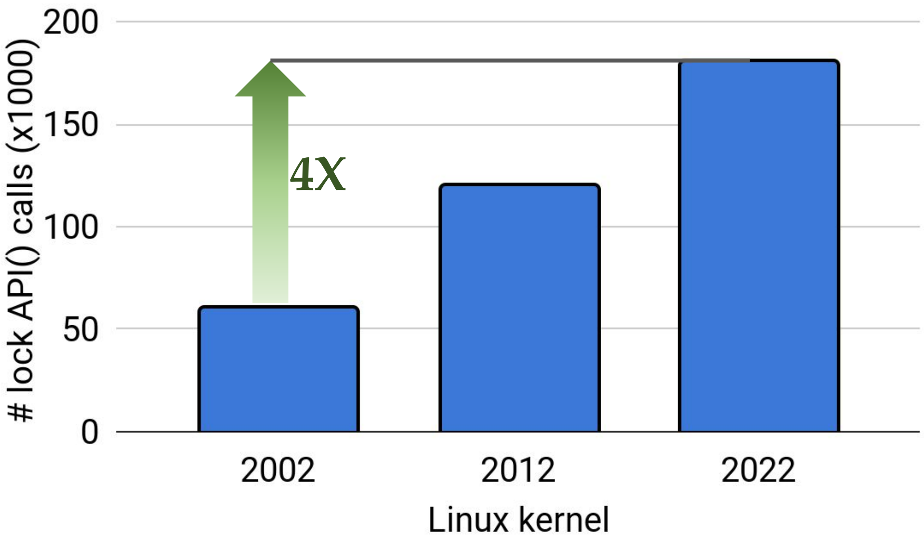
Ship your Critical Section Not Your Data: Enabling Transparent Delegation with TCLocks

Vishal Gupta Kumar Kartikeya Dwivedi Yugesh Kothari
Yueyang Pan Diyu Zhou Sanidhya Kashyap

EPFL

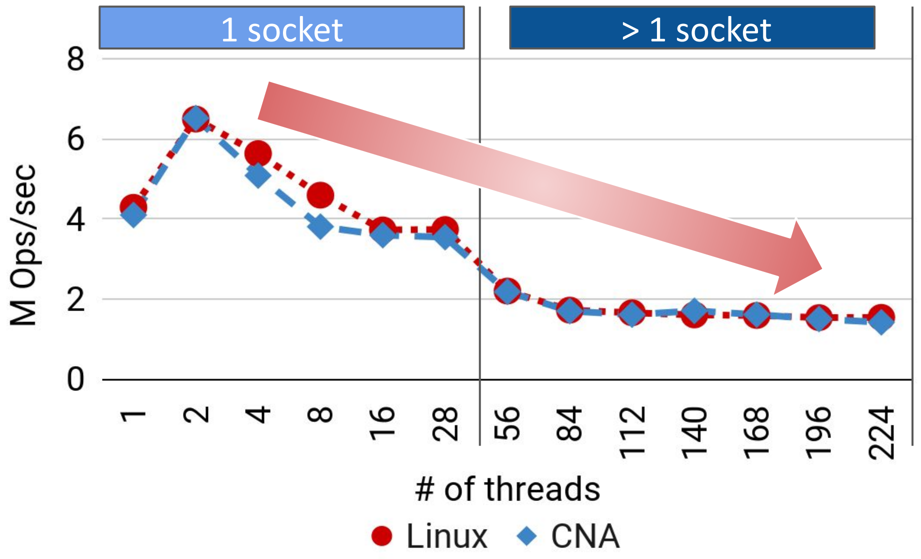
Locks: MOST WIDELY used mechanism



More locks are in use to improve OS scalability

Performance: Micro-benchmark

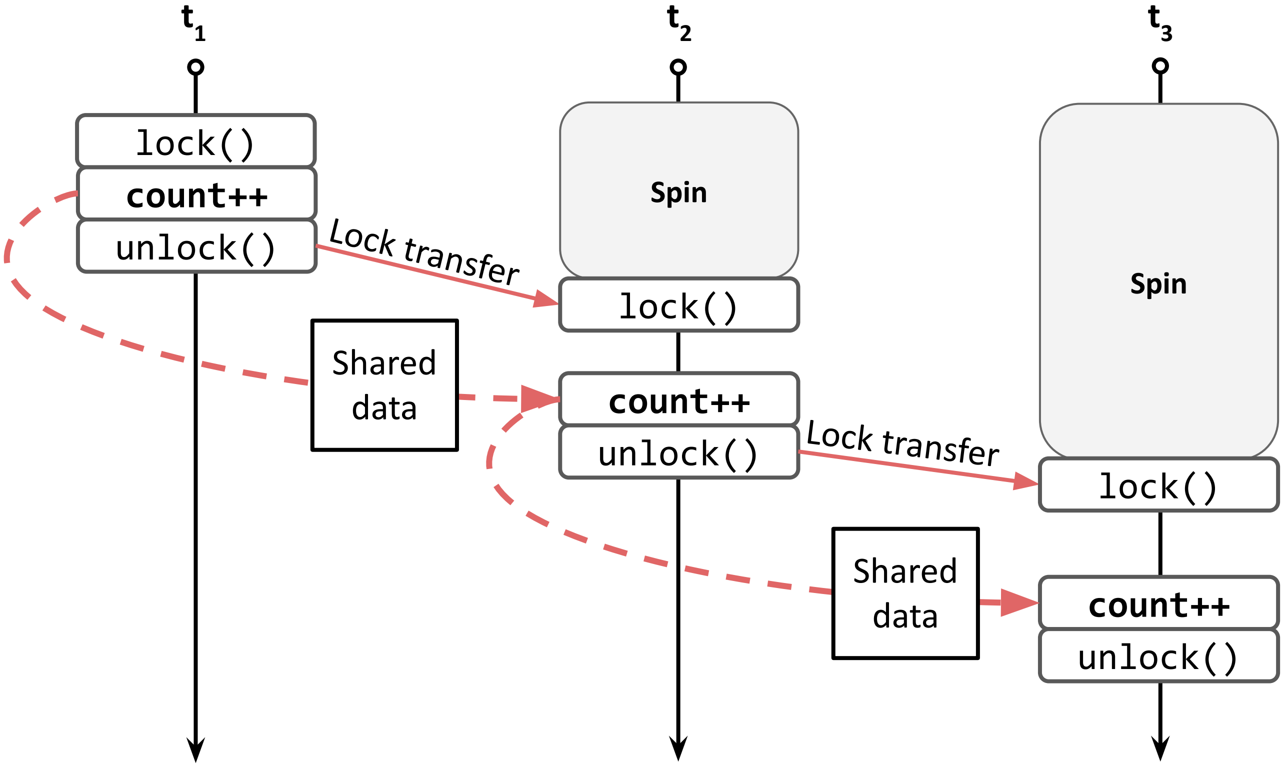
Benchmark: Each thread renames a file in a directory, serialized by a directory lock



- Performance decreases with increasing core count
- NUMA-aware locks (CNA) follow a similar trend

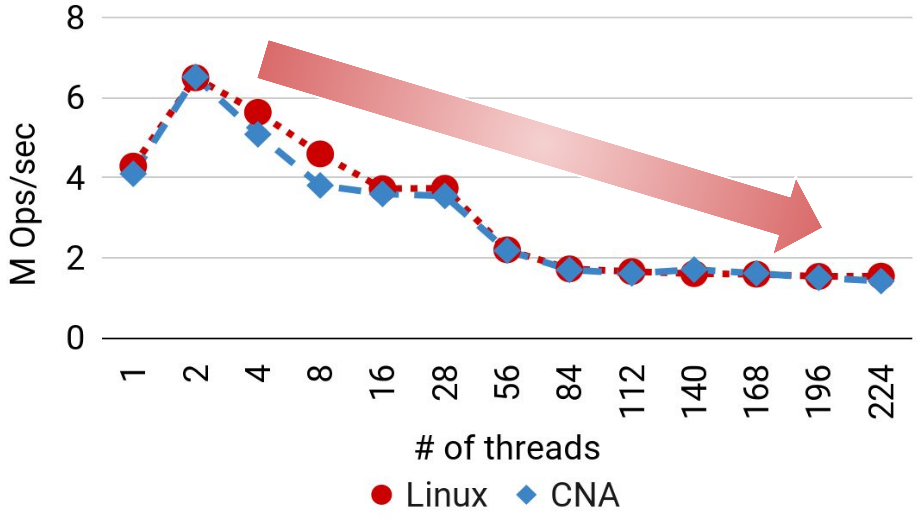
Setup: 8-socket/224-core machine

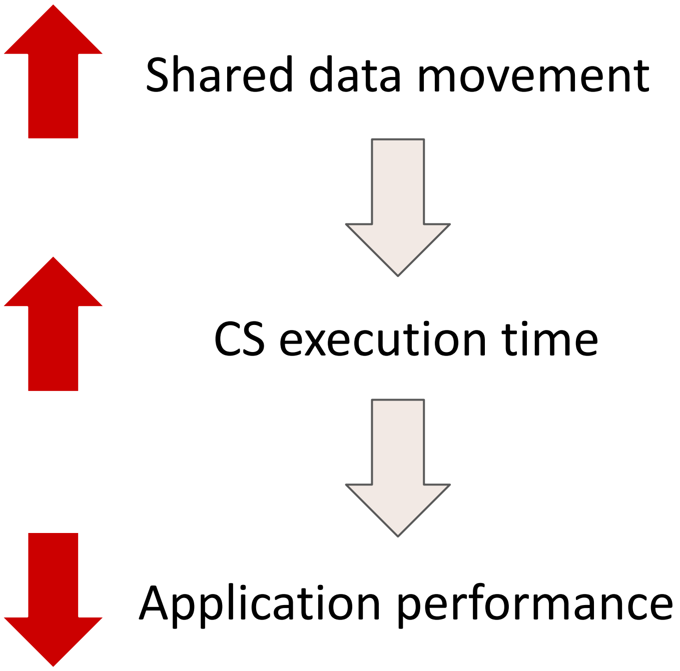
Traditional lock design: Large data movement



t_i: thread i

Traditional lock design: Not ideal





Delegation-style locks

- Similar to a server-client model
  - Server: Lock holder
  - Client: Waits to acquire the lock

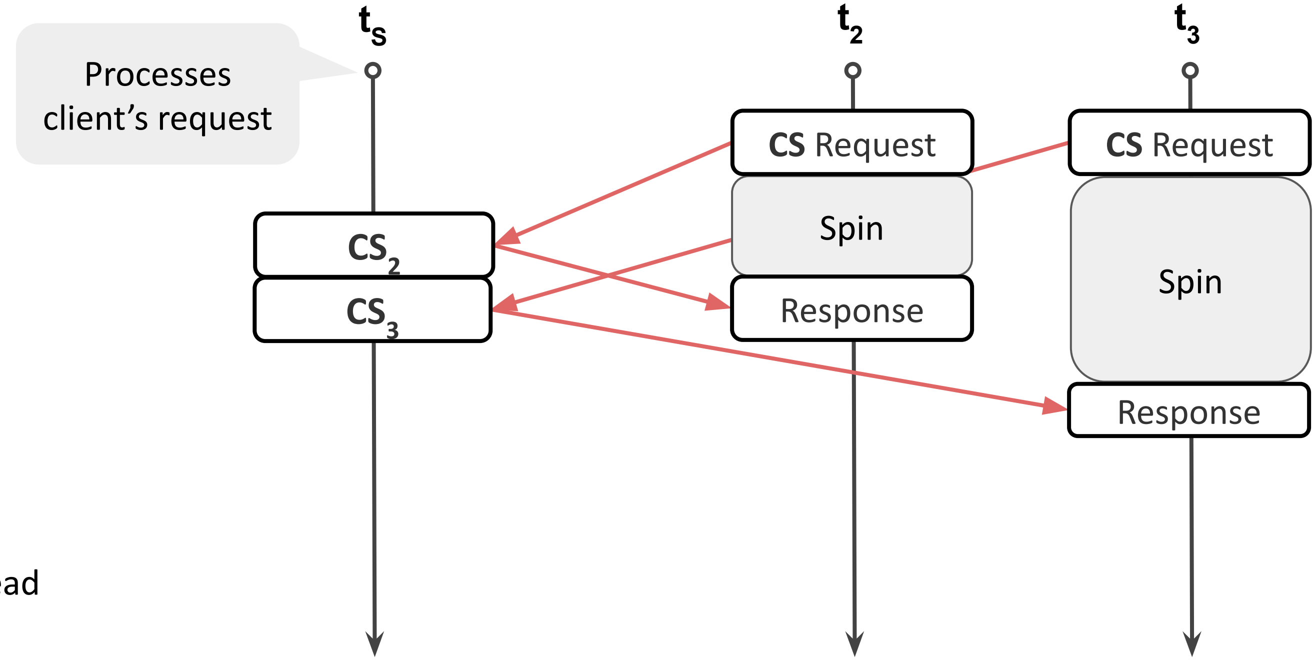
- Client ships its critical section request in the form of a function to the server thread

```
lock()
count++
unlock()
```

```
void incr_func() =
    count++

send_req_to_server(&incr_func)
```

Delegation-style locks



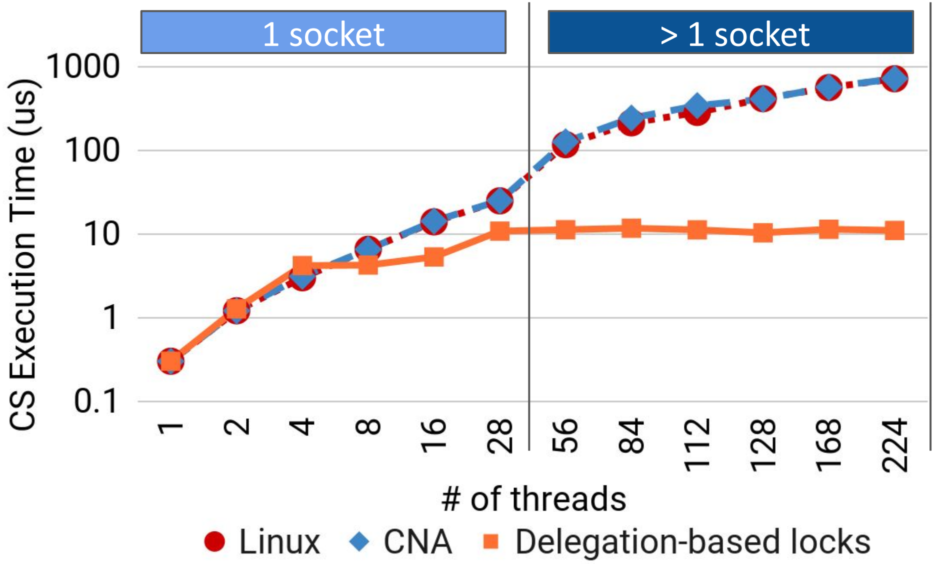
t_s: server thread

t_i: thread i

CS: critical section

Delegation-style locks

Benchmark: Each thread renames a file in a directory, serialized by a directory lock



CS execution time similar with increasing core count

- Minimal shared data movement

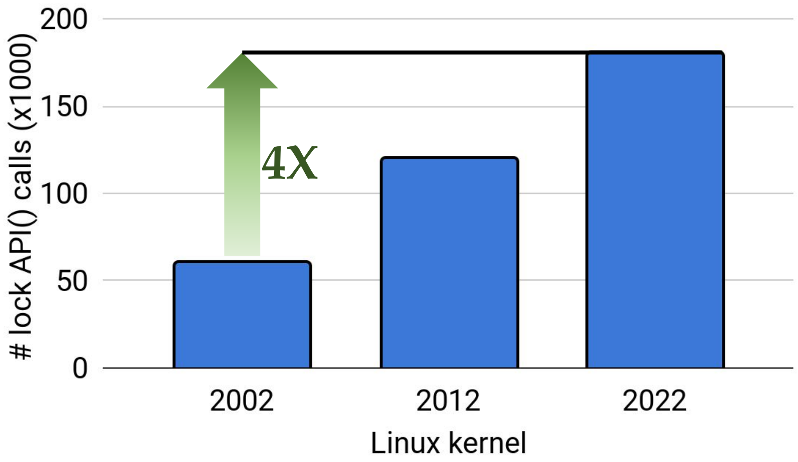
Setup: 8-socket/224-core machine

Delegation locks require code rewrite

```
lock()
count++
unlock()
```

```
void incr_func() =
     count++

send_req_to_server(&incr_func)
```



Existing delegation-based design is impractical for Linux

TCLocks: Goals

- Transparency
  - Use standard lock/unlock APIs without rewriting applications

- Delegation
  - Minimal shared data movement

Transparent delegation

Agenda

- Motivation
- <u>TCLock Design</u>
- TCLock in Linux
- Evaluation
- Discussion

How to achieve transparent delegation?

- **How to capture the thread's context?**
  - Without application rewrite

- **Where to capture the thread's context?**
  - Such that only critical section is captured

- **Does the waiter's thread modify its context?**
  - While the server is executing waiter's critical section

Key idea: Transparent delegation

- **How to capture the thread's context?**
  - Instruction pointer + stack pointer + general-purpose registers

- **Where to capture the thread's context?**
  - Start and end of lock/unlock API

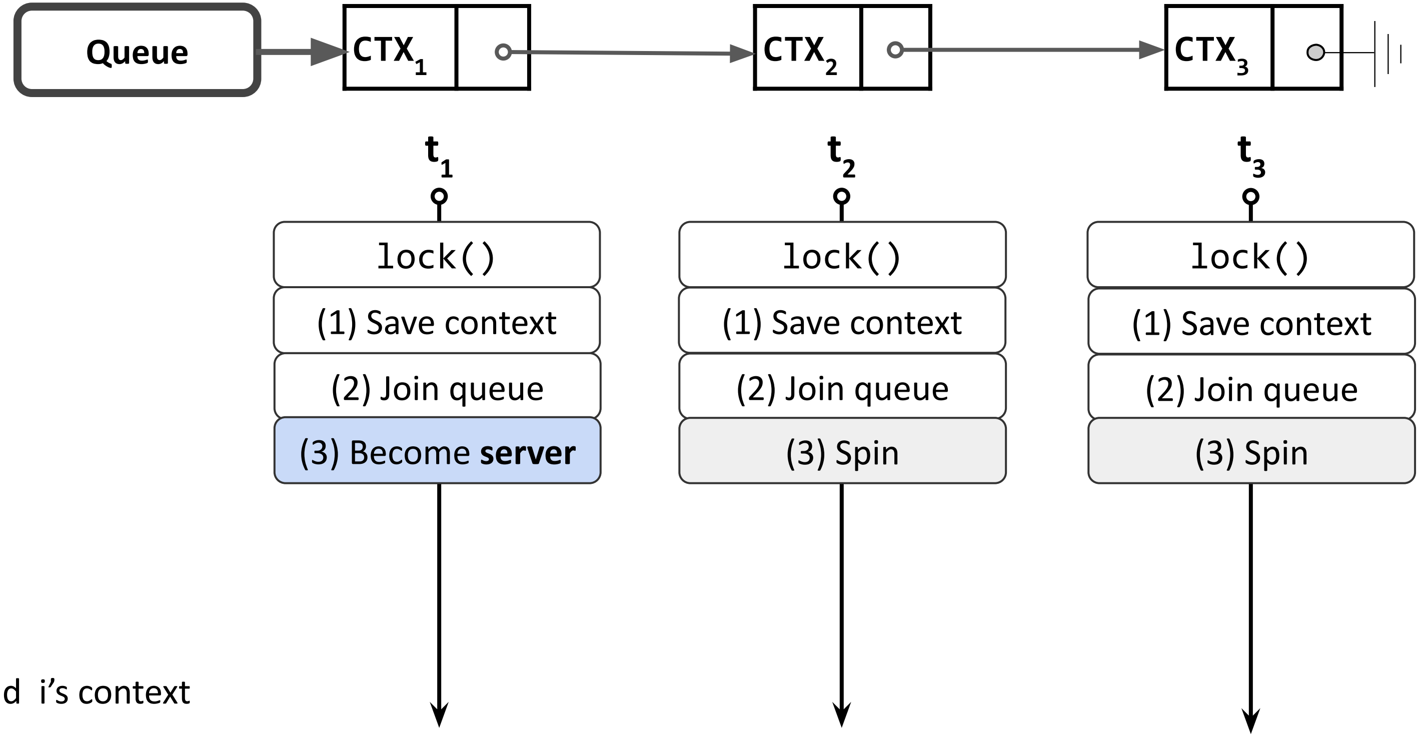
- **Does the waiter's thread modify its context?**
  - No, lock waiter busy waits to acquire the lock

TCLocks: Putting it all together

- Queue-based lock (Similar to qspinlock)
  - List of waiters maintained as a queue
  - Supports locking same lock in different contexts (Task, IRQs, NMI)
- Same `lock/unlock` API

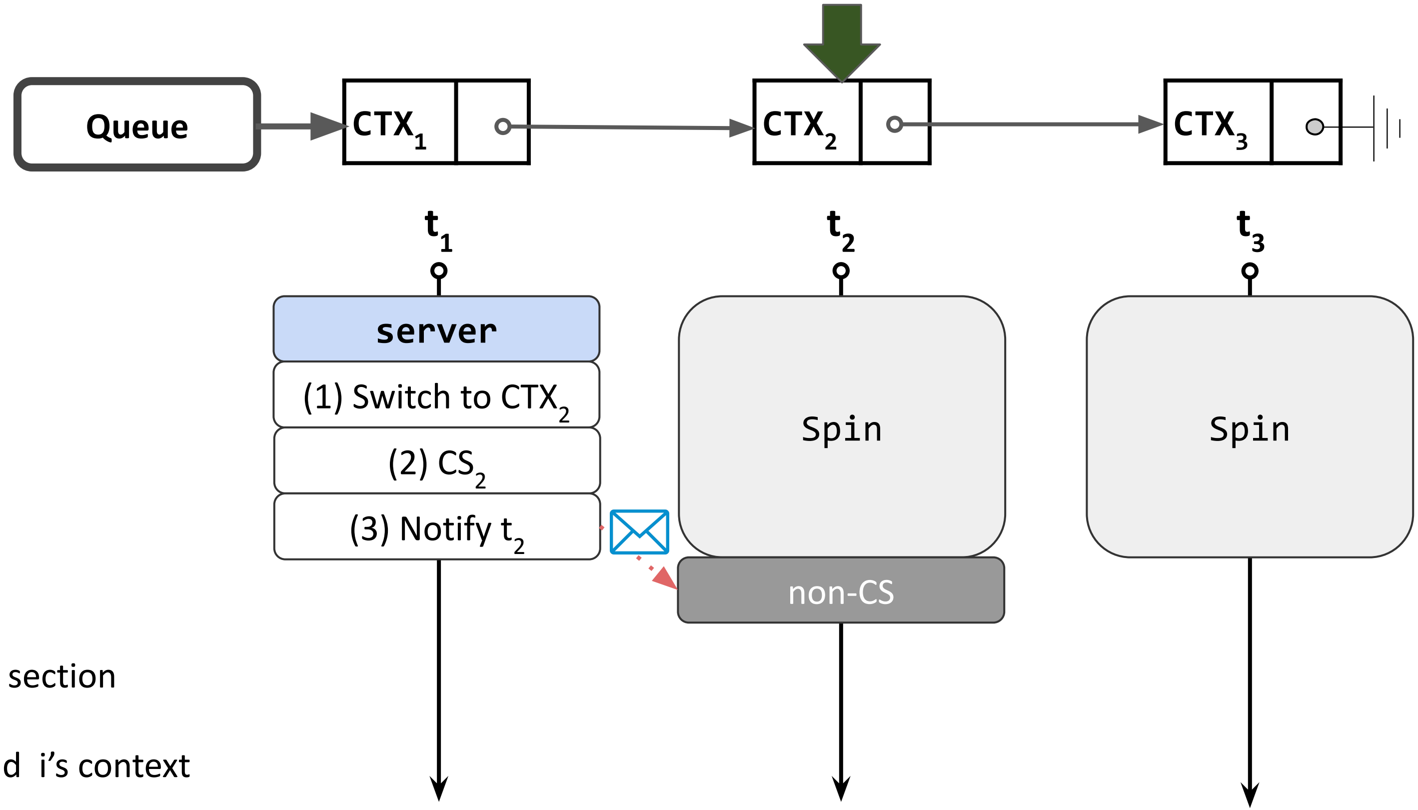
- Server thread batches each waiters' request
- No dedicated server thread
  - Head of the queue becomes the server
  - The role is transferred to the next waiter after some threshold (Batch count)

TCLocks in action: Phase 1



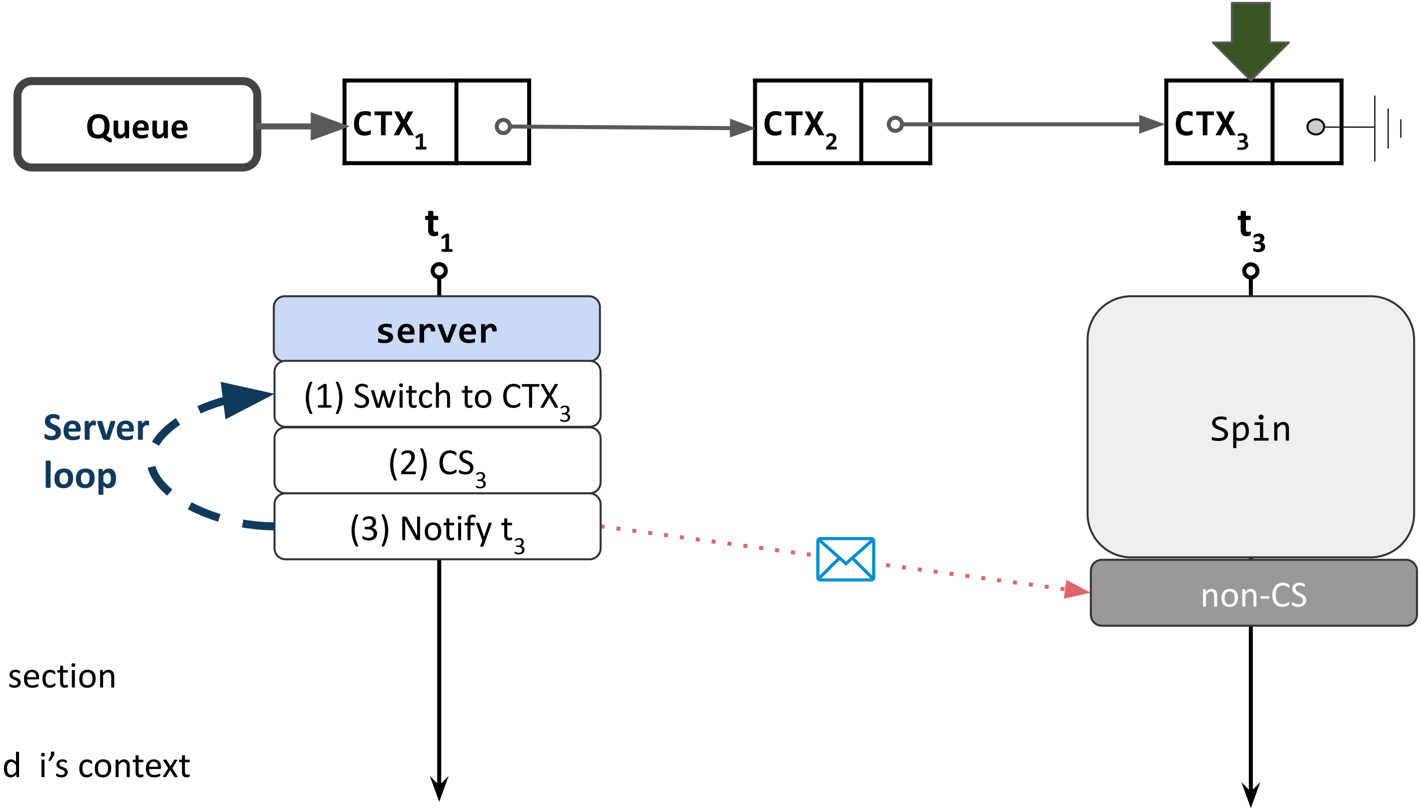
t_i: thread i
CTX_i: thread i's context

TCLocks in action: Phase 2



CS: Critical section
t_i: thread i
CTX_i: thread i's context

TCLocks in action: Phase 2



CS: Critical section
t_i: thread i
CTX_i: thread i's context

Agenda

- Motivation
- TCLock Design
- TCLock in Linux
- Evaluation
- Discussion

TCLocks in Linux

- How to handle:
  - Waiter thread's state modification ?
  - per-CPU variables ?
  - Nested locking ?
  - Out-of-order unlocking ?
  - Mutex ?
  - Reader-Writer Semaphore ?

TCLocks: Waiter’s thread state modification

- Ideal scenario
  - Waiter’s thread does not modify its context
- Reality
  - External events can modify waiter’s context
    - Interrupts: Require stack access
    - Waiter’s parking/wakeup mechanism: Require stack access

- **Solution: Ephemeral stack**
  - An empty piece of memory used only during critical section execution
  - Waiter’s thread switches to Ephemeral stack after saving its context
  - This handles:
    - Interrupts on waiter’s CPU
    - Waiter’s thread parking/wakeup mechanism

TCLocks: per-CPU Variables

- Kernel Assumption:
  - per-CPU variables are stable inside critical section
- With TCLock
  - Critical section running on different CPU.
  - Different per-CPU variables are accessed.
  - Is this behavior correct ?

```c
void mlock_drain_local(void)
{
        struct folio_batch *fbatch;

        local_lock(&mlock_fbatch.lock);
        fbatch = this_cpu_ptr(&mlock_fbatch.fbatch);
        if (folio_batch_count(fbatch))
                mlock_folio_batch(fbatch);
        local_unlock(&mlock_fbatch.lock);
}
```

- **Yes, as long as it runs in a certain context**
  - ! (irqs_disabled() || current->migration_disabled) -> Run Combiner
  - Otherwise, fallback to qspinlock

https://elixir.bootlin.com/linux/v6.6/source/mm/mlock.c#L217

TCLocks: Nested Locking

- Kernel Assumption:
  - Multiple different locks can nest with arbitrary depth.
  - Same lock can also nest in different execution contexts.
- With TCLock
  - Server thread can become a waiter thread for nested lock

```
static void __d_move(struct dentry *dentry, struct dentry *target,
                     bool exchange)
        /* target is not a descendent of dentry->d_parent */
        spin_lock(&target->d_parent->d_lock);
        spin_lock_nested(&old_parent->d_lock, DENTRY_D_LOCK_NESTED);
} else {
        BUG_ON(p == dentry);
        spin_lock(&old_parent->d_lock);
        if (p != target)
                spin_lock_nested(&target->d_parent->d_lock,
                                DENTRY_D_LOCK_NESTED);
}
spin_lock_nested(&dentry->d_lock, 2);
spin_lock_nested(&target->d_lock, 3);
```

- **Solution similar to interrupt processing mechanism**
  - Save server thread's context on the stack before calling the nested lock.
  - Restore the server thread's context when nested lock returns.

https://elixir.bootlin.com/linux/v6.6/source/fs/dcache.c#L2962

TCLocks: Out-of-order Unlocking

- Kernel Assumption:
    - Nested locks can be unlocked in any order
- With TCLock
    - Server thread returns to its own context in the unlock function.
    - It can return before the lock it held is not unlocked

```c
/*
 * Splice contents of ipipe to opipe.
 */
static int splice_pipe_to_pipe(struct pipe_inode_info *ipipe,
                               struct pipe_inode_info *opipe,
                               size_t len, unsigned int flags)

        pipe_double_lock(ipipe, opipe);

        pipe_unlock(ipipe);
        pipe_unlock(opipe);
```

- **Solution: Use an array to track lock order**
    - Delay unlocking the out-of-order unlocked lock until the remaining locks are unlocked.

https://elixir.bootlin.com/linux/v6.6/source/fs/splice.c#L1673

TCLocks: Mutex

- Differences from Spinlock:
    - Server thread state is stored in task_struct instead of per-CPU variables.
- Rest is similar to mutex in the kernel:
    - Except, currently it doesn't support: Mutex_lock_interruptable / mutex_lock_killable.
    - It is handled same as mutex_lock.

TCLocks: Reader-writer semaphore

- Phase-based reader-writer lock:
  - Reader phase allows all readers to proceed, while writers are waiting.
  - Writer phase combines all writers using a server thread, while readers are waiting.

Agenda

- Motivation
- TCLock Design
- TCLock in Linux
- <u>Evaluation</u>
- Discussion

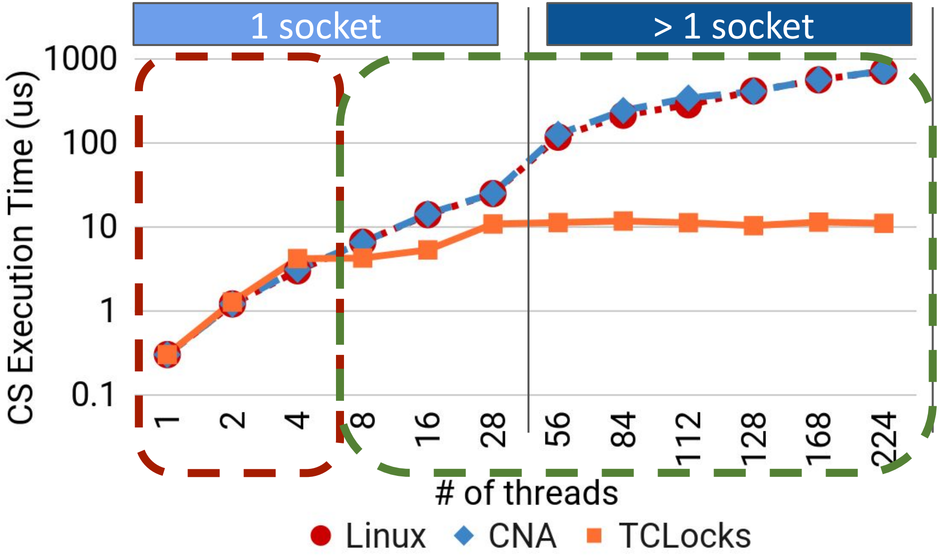
TCLocks: Evaluation

- Does TCLocks reduce the time spent in critical section?
- Does TCLocks improve application performance?

Hardware: 8-socket/224-core Intel machine

Evaluation: CS execution time

Benchmark: Each thread renames a file in a directory, serialized by a directory lock

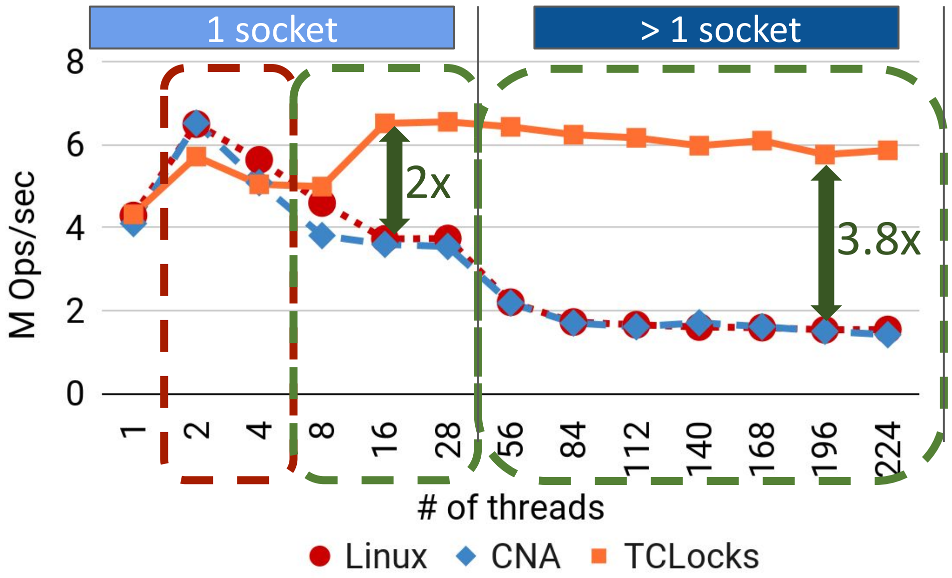


- **> 4 threads**
  - Minimal shared data movement
- **≤ 4 threads**
  - Context-switch overhead
  - Not enough batching

Setup: 8-socket/224-core machine

Evaluation: Micro-benchmark

Benchmark: Each thread renames a file in a directory, serialized by a directory lock

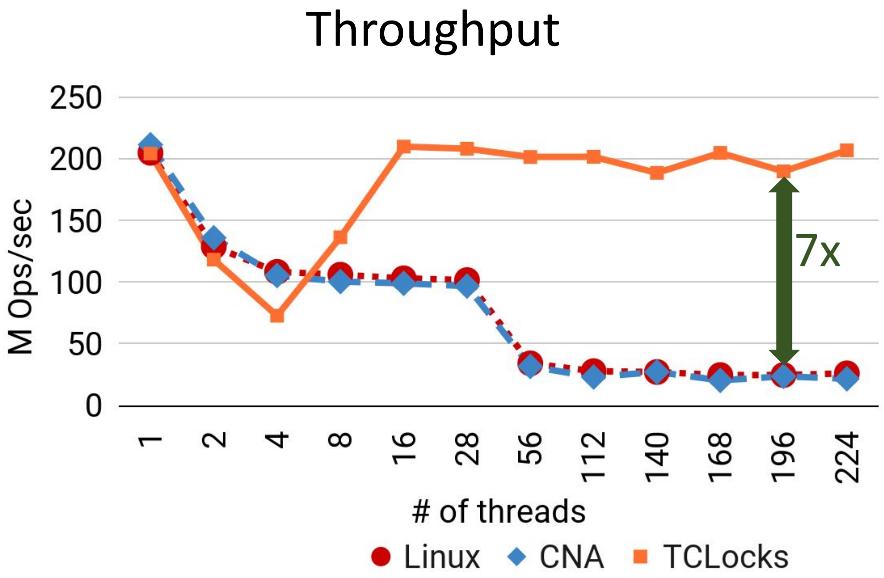


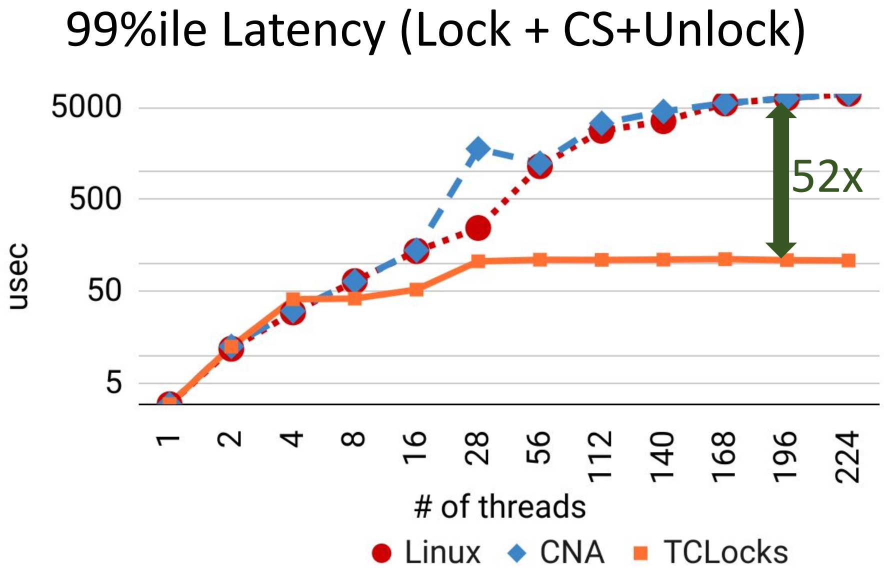
- **Within a socket:**
  - Minimal shared data movement
- **Across socket:**
  - NUMA-aware policy
- **2 - 4 cores:**
  - Context-switch overhead
  - Not enough batching

Setup: 8-socket/224-core machine

Evaluation: Throughput and Latency

Benchmark: Each thread update an entry in hash-table, serialized by global spinlock





TCLocks provides better throughput with lower 99% latency.

Agenda

- Motivation
- TCLock Design
- TCLock in Linux
- Evaluation
- Discussion

Discussion questions

- How to set the batching count ?
  - Throughput vs Latency
- How to handle performance regression at low contention (2-4 threads) ?
  - Switch between different lock mechanisms
  - TCLocks already uses qspinlock for certain contexts (IRQs disabled) and combining for others.
- How to handle CPU time accounting for server thread ?
  - Server thread might eat up the CPU time while executing other waiter’s critical section.
  - Problem similar to CPU time accounting for interrupt processing.

Discussion questions

- How to provide **current** macro correctly within and outside the critical section ?
  - Within a critical section, we need current of waiter’s thread on server CPU.
  - Outside the critical section, we need current of server thread on server CPU.

Conclusion

- Existing lock design:
  - Traditional lock design has more shared data movement
  - Delegation-based lock design requires application modification
- **TCLocks**: Provides transparent delegation
  - Capture thread's context at right time
- Key takeaway:
  - Applications can now use delegation-style locks without modification

https://rs3lab.github.io/TCLocks/

Thank you!

Backup slides

TCLocks: Pseudo-code

```
spin_lock ():
    node = get_per_cpu_node()
    save_context_on_node(node)
    join_queue(node)
    if( not head_of_queue() ):
        While node.wait is True:
            Continue
        restore _context_from_node(node)
        Return
    while True:
        qnext = get_next_thread()
        switch_to(qnext)
        notify(qnext)
        If ( batch_count_exceeded() ); break
```

```
spin_unlock():
    If (server_context() ):
        switch_to(server)
    else
        Lock = Unlocked
```