# Linux Synchronization Mechanism: spinlock

Adrian Huang | Dec, 2022

*** Based on kernel 5.11 (x86_64) – QEMU**
*** 2-socket CPUs (4 cores/socket)**
*** 16GB memory**
*** Kernel parameter: nokaslr norandmaps**
*** KASAN: disabled**
*** Userspace: ASLR is disabled**
*** Legacy BIOS**

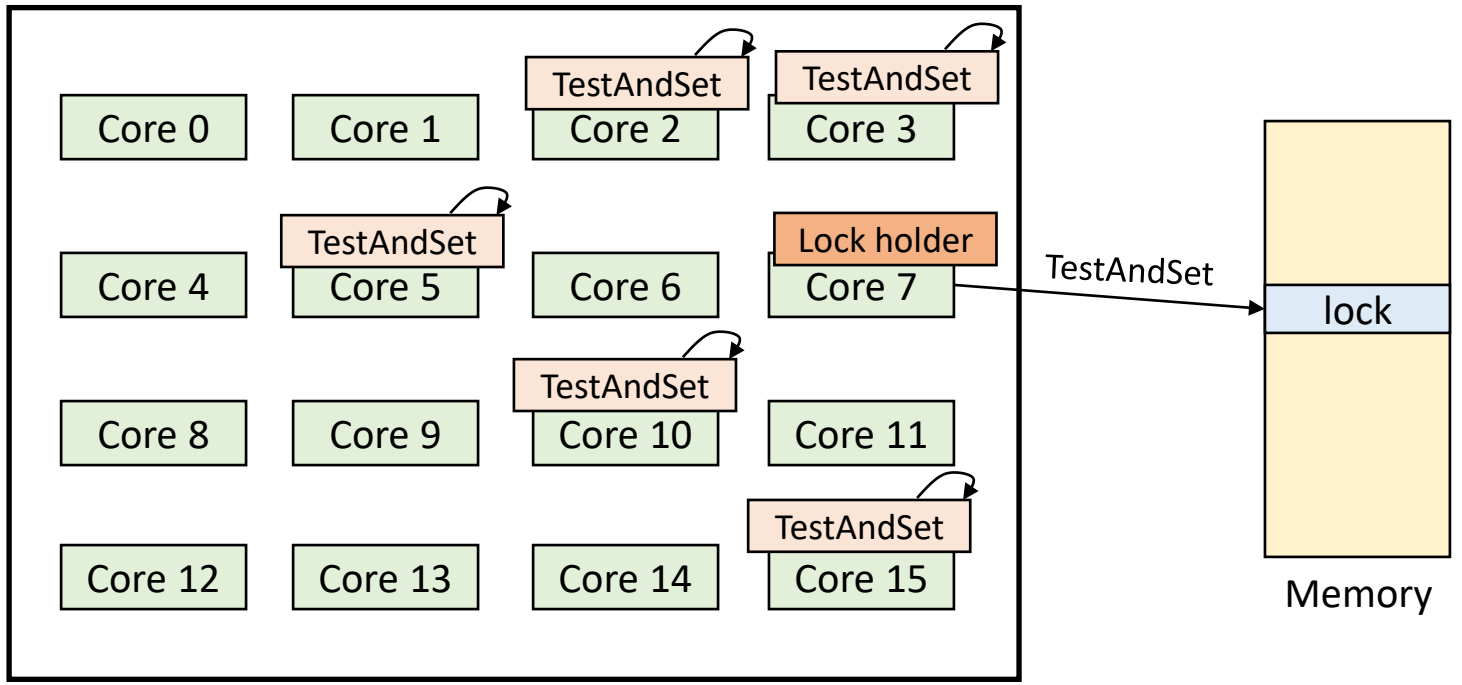
# Agenda

- Spinlock history – Approach evolution
  - ✓Simple Approach: Spin on test-and-set
  - ✓Test and test-and-set (spin on read)
  - ✓Ticket spinlock
  - ✓MCS (Mellor-Crummey & Scott) lock
    - Performance benchmark: Ticket spinlock vs MCS lock
    - MCS Lock History in Linux Kernel
- Current spinlock approach in Linux kernel: qspinlock (Queue spinlock)
- spin_lock() SMP & UP
- spin_lock() API variants
  - ✓How to use those variants in different scenarios
- Spinlock derivative: rwlock and seqlock

# Simple Approach: Spin on test-and-set

```
Init      lock := CLEAR;
Lock      while (TestAndSet (lock) = BUSY);
Unlock    lock := CLEAR;
```

Test-and-set (atomic):
1. old_value = read a memory location
2. Write 1 to a memory location
3. Return old_value

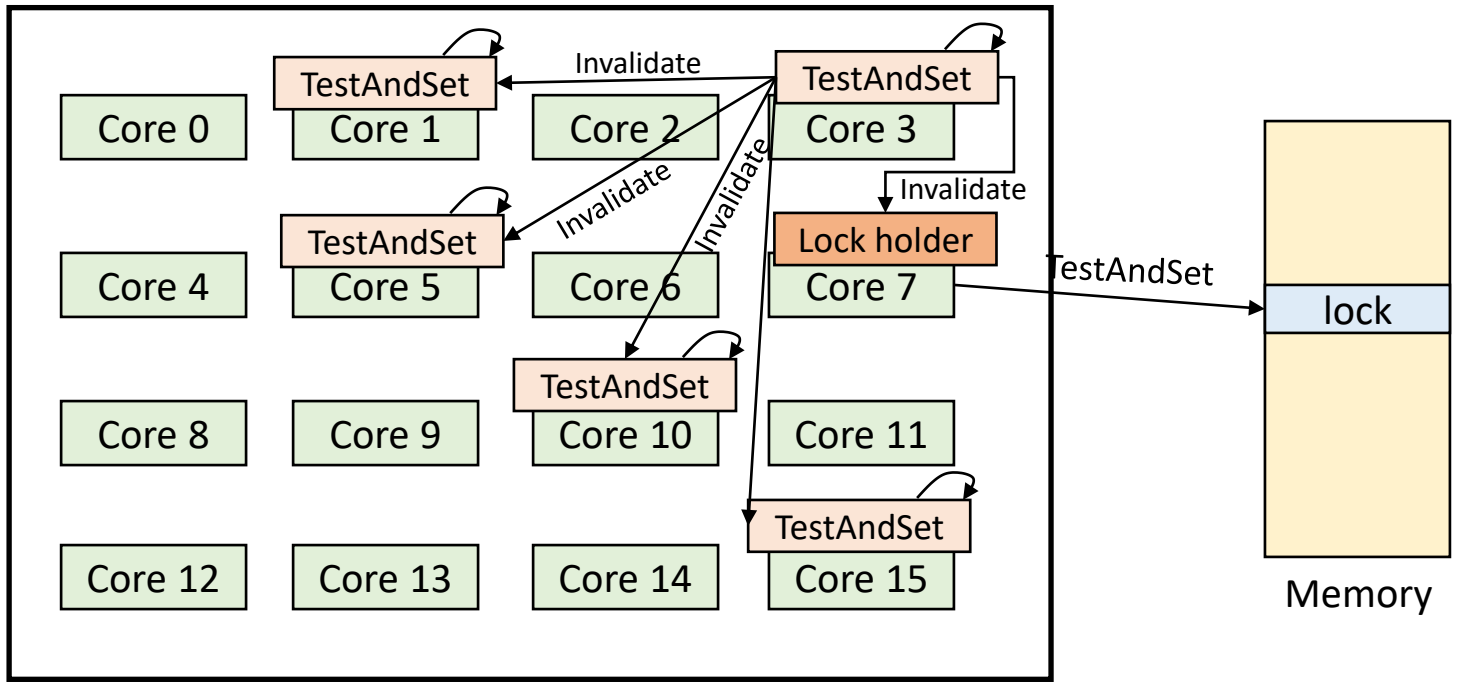



**[Spinning Cores] Keep consuming memory bus (write 1): Cache coherence**

# Simple Approach: Spin on test-and-set

```
Init      lock := CLEAR;
Lock      while (TestAndSet (lock) = BUSY);
Unlock    lock := CLEAR;
```

Test-and-set (atomic):
1. old_value = read a memory location
2. Write 1 to a memory location
3. Return old_value



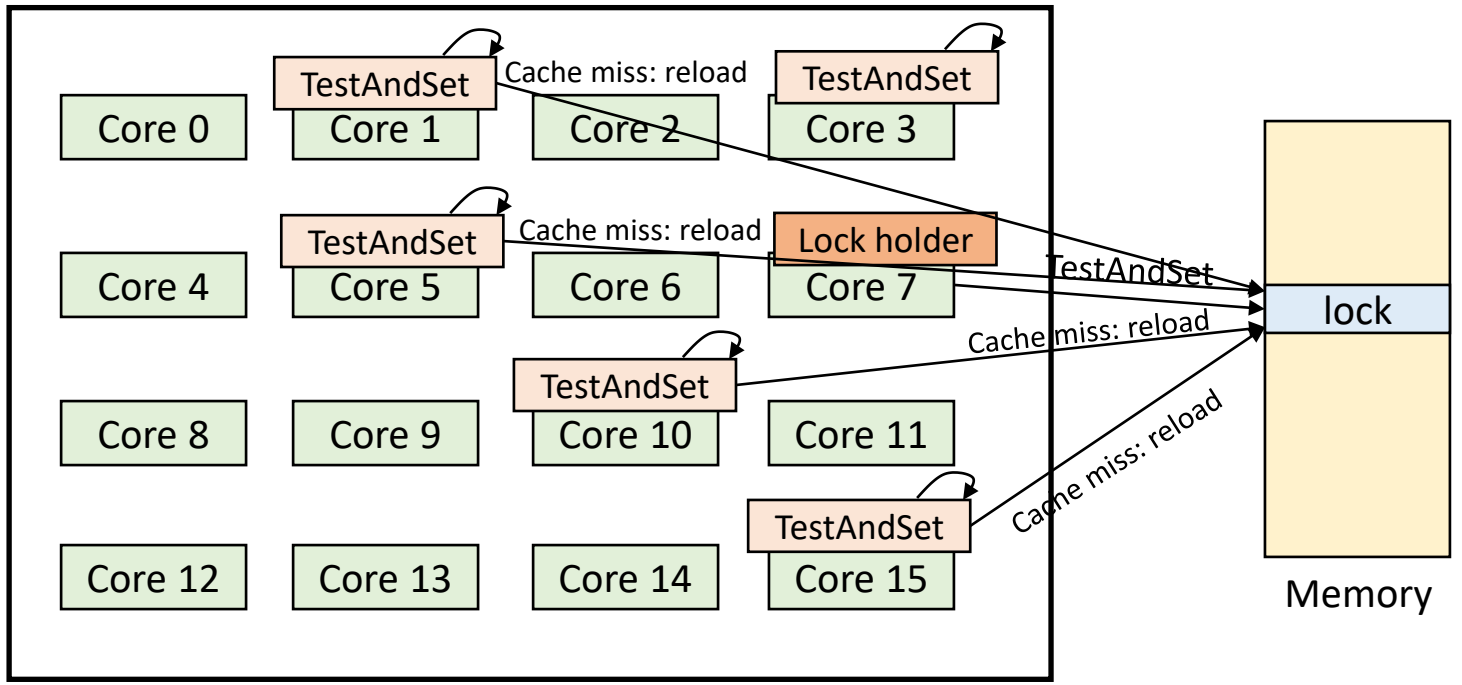

- **Read Invalidate** message of MESI Protocol Messages
  - ✓ Reference: C.2.2 MESI Protocol Messages of Is Parallel Programming Hard, And, If So, What Can You Do About It?

**Due to the memory write, spinning cores invalidate cache copies of cores even if the value is not changed**

# Simple Approach: Spin on test-and-set

```
Init        lock := CLEAR;
Lock        while (TestAndSet (lock) = BUSY);
Unlock      lock := CLEAR;
```

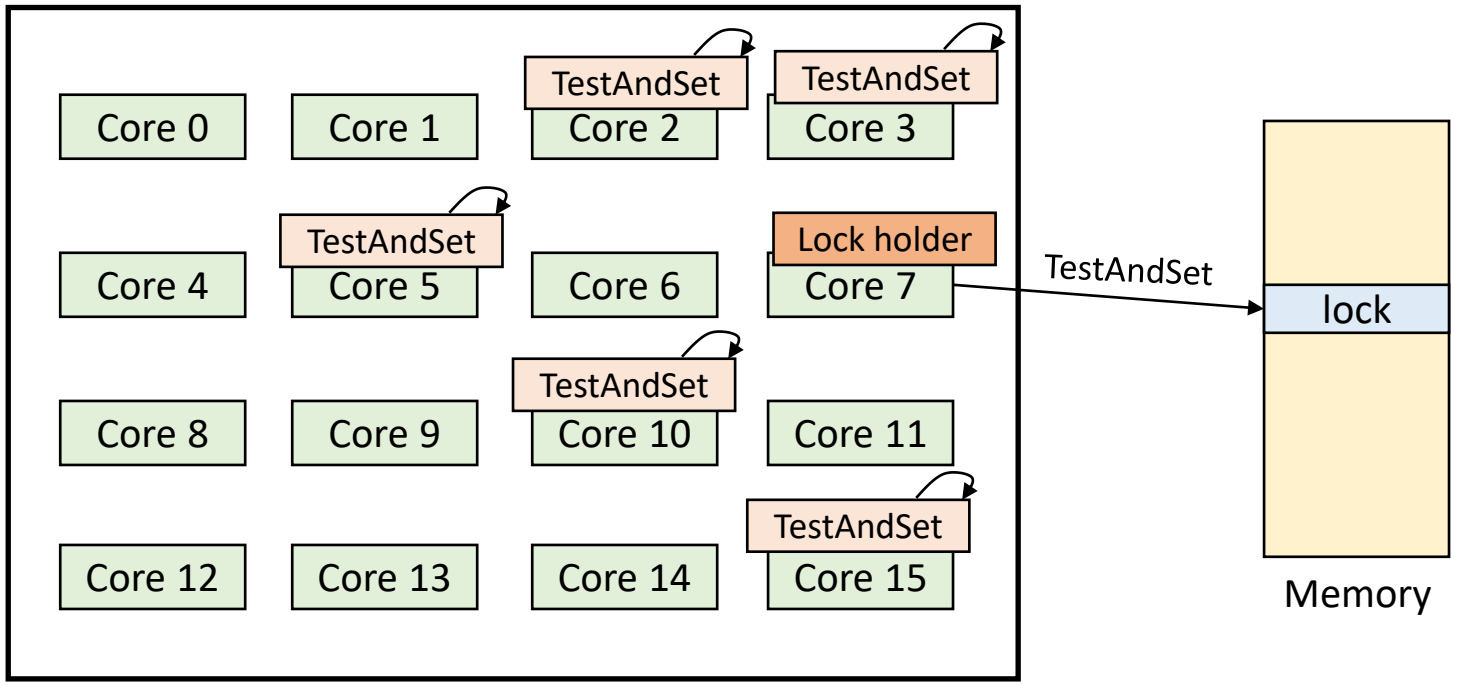
Test-and-set (atomic):
1. old_value = read a memory location
2. Write 1 to a memory location
3. Return old_value




**[Spinning Cores] Cores reload the memory due to the cache miss: performance impact**

# Test and test-and-set (spin on read)

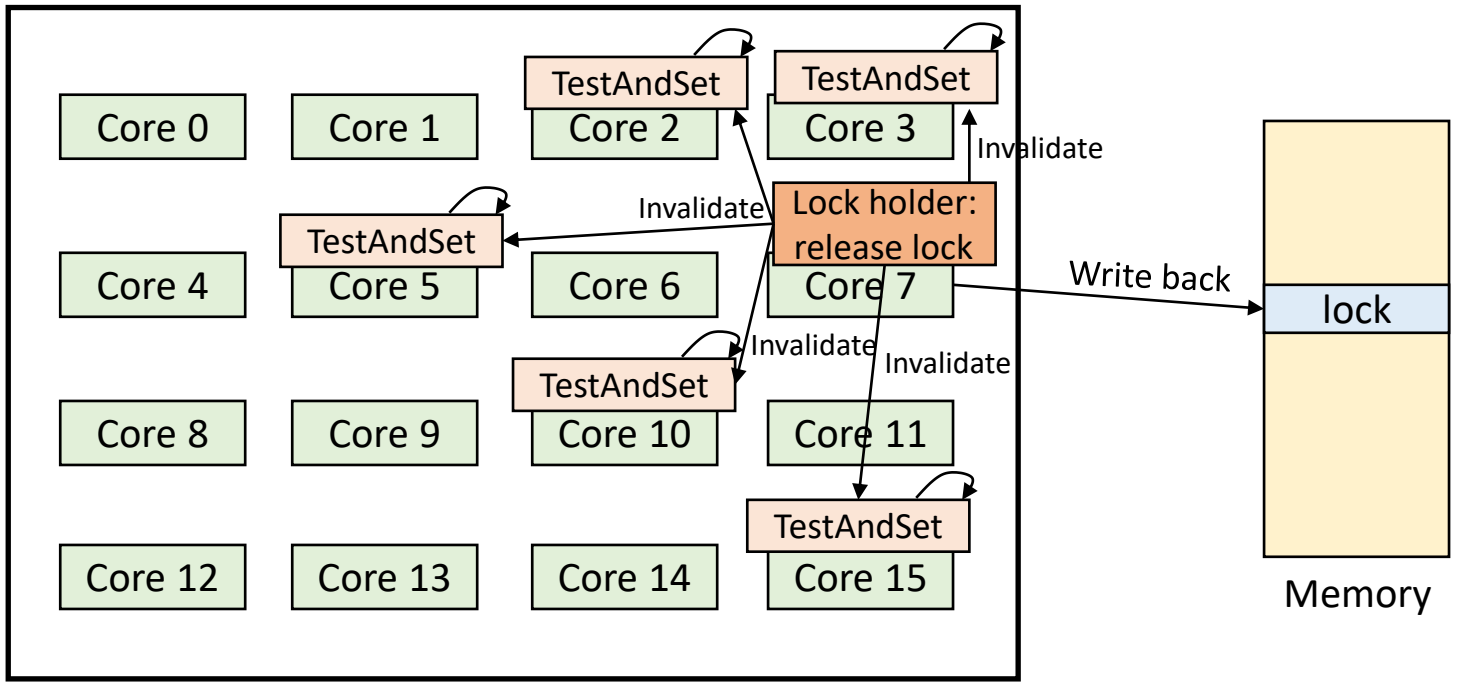
```
Lock        while (lock = BUSY or TestAndSet (lock) = BUSY)
```




1. **Spinning is done in the cache without consuming memory bus**
2. **Reduce the repeated test-and-set cost if the lock is held**

# Test and test-and-set (spin on read)

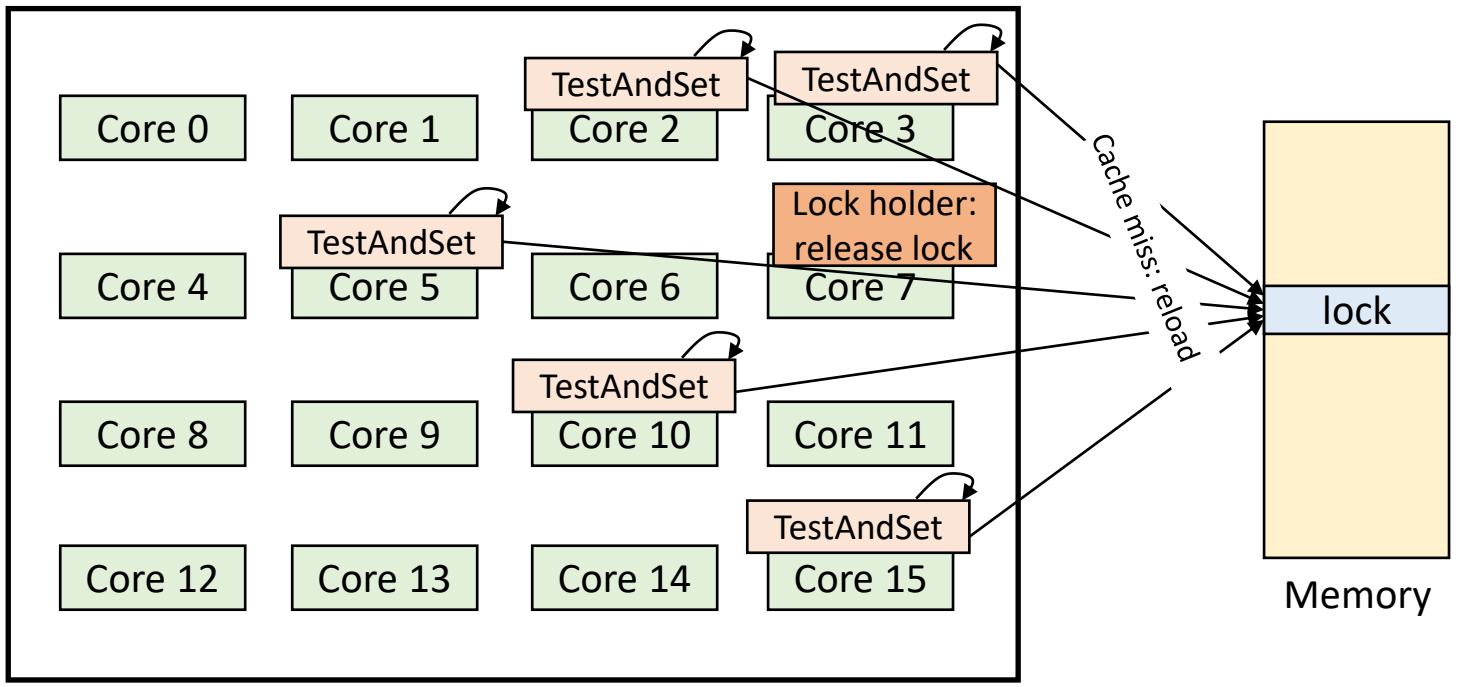
```
Lock        while (lock = BUSY or TestAndSet (lock) = BUSY)
```




1. **Spinning is done in the cache without consuming memory bus**
2. **Reduce the repeated test-and-set cost if the lock is held**

# Test and test-and-set (spin on read)

```
Lock        while (lock = BUSY or TestAndSet (lock) = BUSY)
```




1. **Spinning cores incur a read miss and fetch the new value back into cache**
2. **Spinning cores compete for accessing memory bus**
3. **The first core to test-and-set will acquire the lock**
4. **Other spinning cores cannot get lock: invalidate caches & cache misses**
   - **Quiescence: These operations are finished**

# Performance Comparison

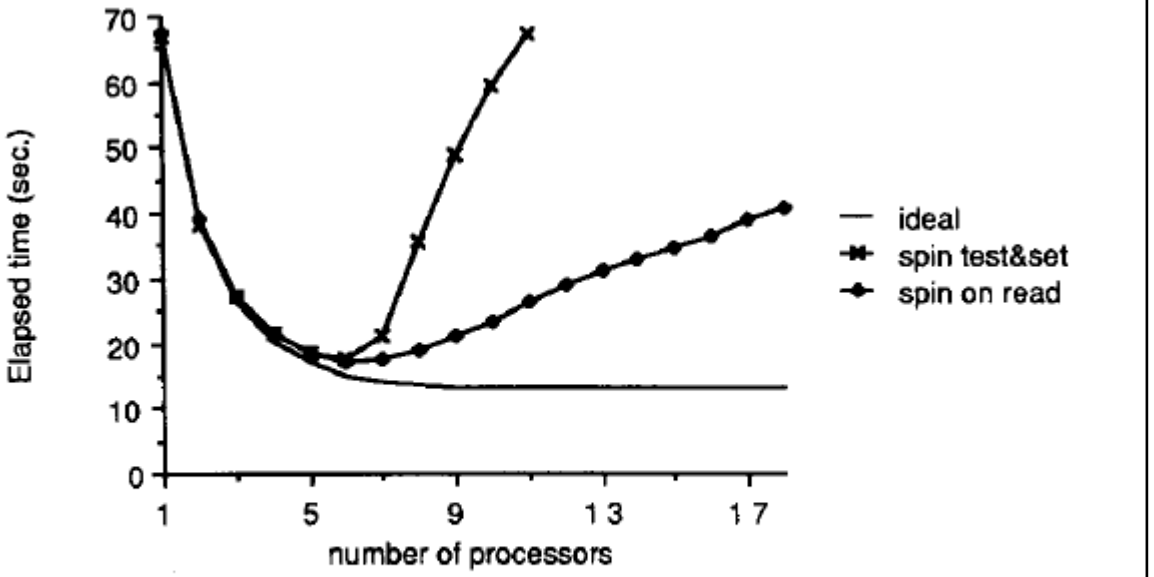



Fig. 1.Principal performance comparison: elapsed time (second) to execute benchmark (measured). Each processor loops one million/$P$ times: acquire lock, do critical section, release lock, and compute.

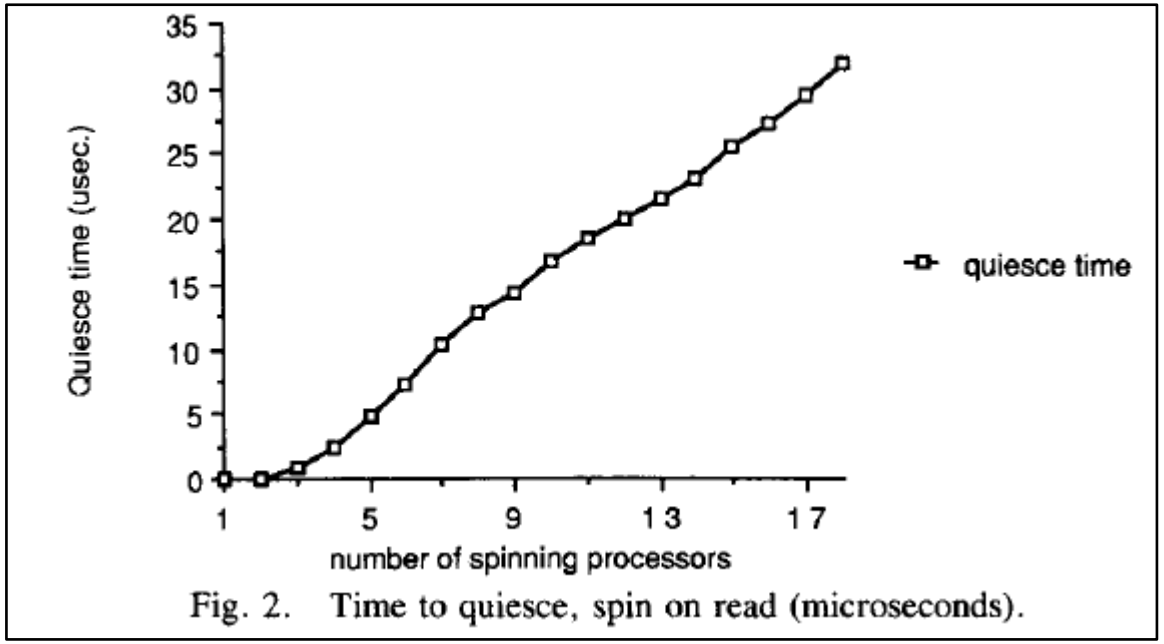



Fig. 2. Time to quiesce, spin on read (microseconds).

Reference from: T. E. Anderson, "**The performance of spin lock alternatives for shared-money multiprocessors**", IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pp. 6-16, 1990.

# Issue Statements about “Spin on test-and-set” and “Test and test-and-set”

- Cache coherence
  - ✓[Cache-line bouncing] Scalability issue
- All spinning cores compete the lock when the lock is freed by lock holder
  - ✓Unfairness
    - ➢Ticket spinlock is proposed to fix the unfairness

# Ticket spinlock (v2.6.25: released in April 2008)

**Concept: pseudo code**

```
struct spinlock_t {
        int current_ticket;
        int next_ticket;
};

void spin_lock(spinlock_t *lock)
{
        int t = atomic_fetch_and_inc(&lock->next_ticket);
        while (t != lock ->current_ticket)
                ; /* spin */
}

void spin_unlock(spinlock_t *lock)
{
        lock ->current_ticket++;
}
```

```
commit 314cdbefd1fd0a7acf3780e9628465b77ea6a836
Author: Nick Piggin <npiggin@suse.de>
Date:   Wed Jan 30 13:31:21 2008 +0100

    x86: FIFO ticket spinlocks

    Introduce ticket lock spinlocks for x86 which are FIFO. The implementation
    is described in the comments. The straight-line lock/unlock instruction
    sequence is slightly slower than the dec based locks on modern x86 CPUs,
    however the difference is quite small on Core2 and Opteron when working out of
    cache, and becomes almost insignificant even on P4 when the lock misses cache.
    trylock is more significantly slower, but they are relatively rare.

    On an 8 core (2 socket) Opteron, spinlock unfairness is extremely noticable,
    with a userspace test having a difference of up to 2x runtime per thread, and
    some threads are starved or "unfairly" granted the lock up to 1 000 000 (!)
    times. After this patch, all threads appear to finish at exactly the same
    time.

    The memory ordering of the lock does conform to x86 standards, and the
    implementation has been reviewed by Intel and AMD engineers.

    The algorithm also tells us how many CPUs are contending the lock, so
    lockbreak becomes trivial and we no longer have to waste 4 bytes per
    spinlock for it.

    After this, we can no longer spin on any locks with preempt enabled
    and cannot reenable interrupts when spinning on an irq safe lock, because
    at that point we have already taken a ticket and the would deadlock if
    the same CPU tries to take the lock again.  These are questionable anyway:
    if the lock happens to be called under a preempt or interrupt disabled section,
    then it will just have the same latency problems. The real fix is to keep
    critical sections short, and ensure locks are reasonably fair (which this
    patch does).

    Signed-off-by: Nick Piggin <npiggin@suse.de>
    Signed-off-by: Thomas Gleixner <tglx@linutronix.de>
    Signed-off-by: Ingo Molnar <mingo@elte.hu>
```

# Ticket spinlock (v2.6.25: released in April 2008)

**Concept: pseudo code**

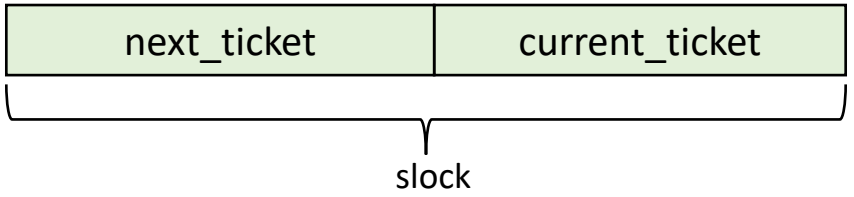
```
struct spinlock_t {
        int current_ticket;
        int next_ticket;
};

void spin_lock(spinlock_t *lock)
{
        int t = atomic_fetch_and_inc(&lock->next_ticket);
        while (t != lock ->current_ticket)
                ; /* spin */
}

void spin_unlock(spinlock_t *lock)
{
        lock ->current_ticket++;
}
```

**v2.6.25 implementation**

```
commit 3a556b26a2718e48aa2b6ce06ea4875ddcd0778e
Author: Nick Piggin <npiggin@suse.de>
Date:   Wed Jan 30 13:33:00 2008 +0100

    x86: big ticket locks

    This implements ticket lock support for more than 255 CPUs on x86. The
    code gets switched according to the configured NR_CPUS.

    Signed-off-by: Nick Piggin <npiggin@suse.de>
    Signed-off-by: Ingo Molnar <mingo@elte.hu>
    Signed-off-by: Thomas Gleixner <tglx@linutronix.de>
```

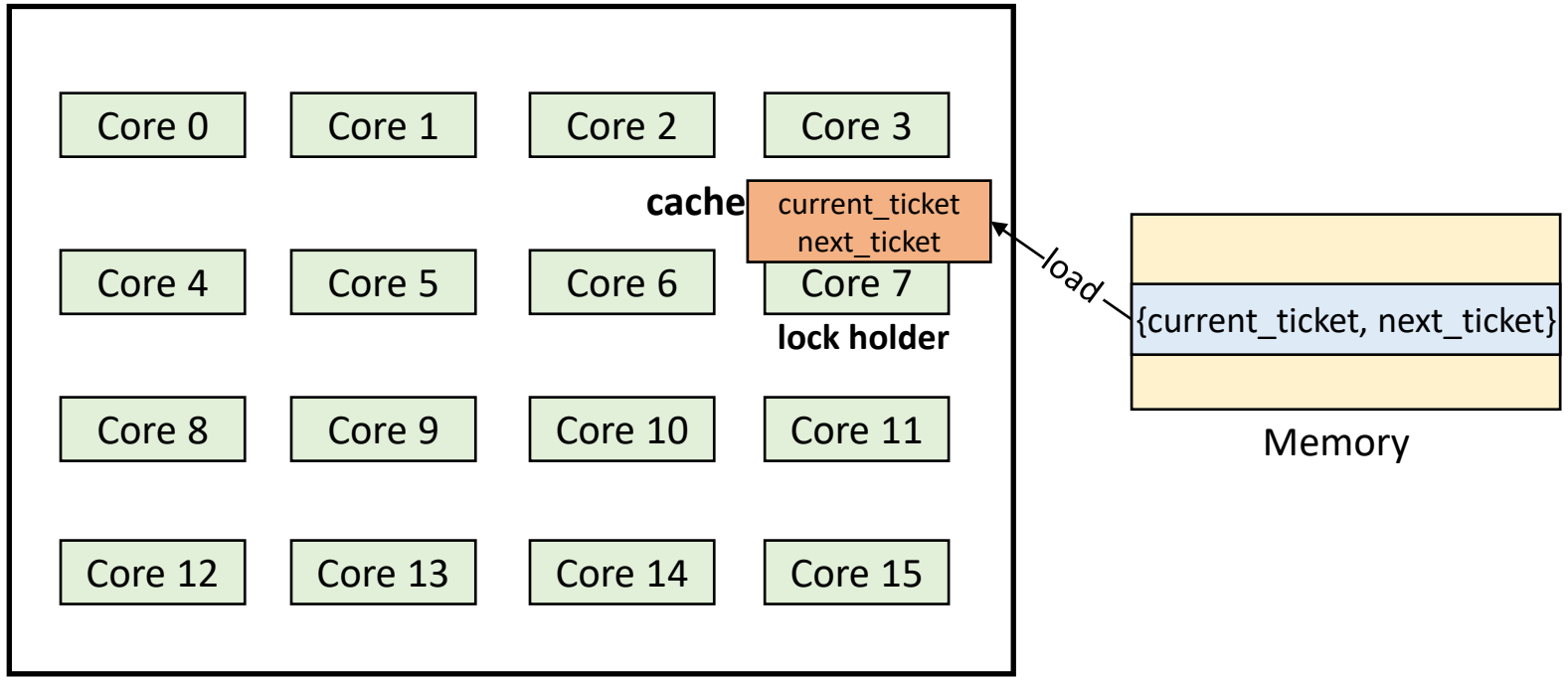
```
typedef struct arch_spinlock {
        unsigned int slock;
} arch_spinlock_t;
```

arch/x86/include/asm/spinlock_types.h



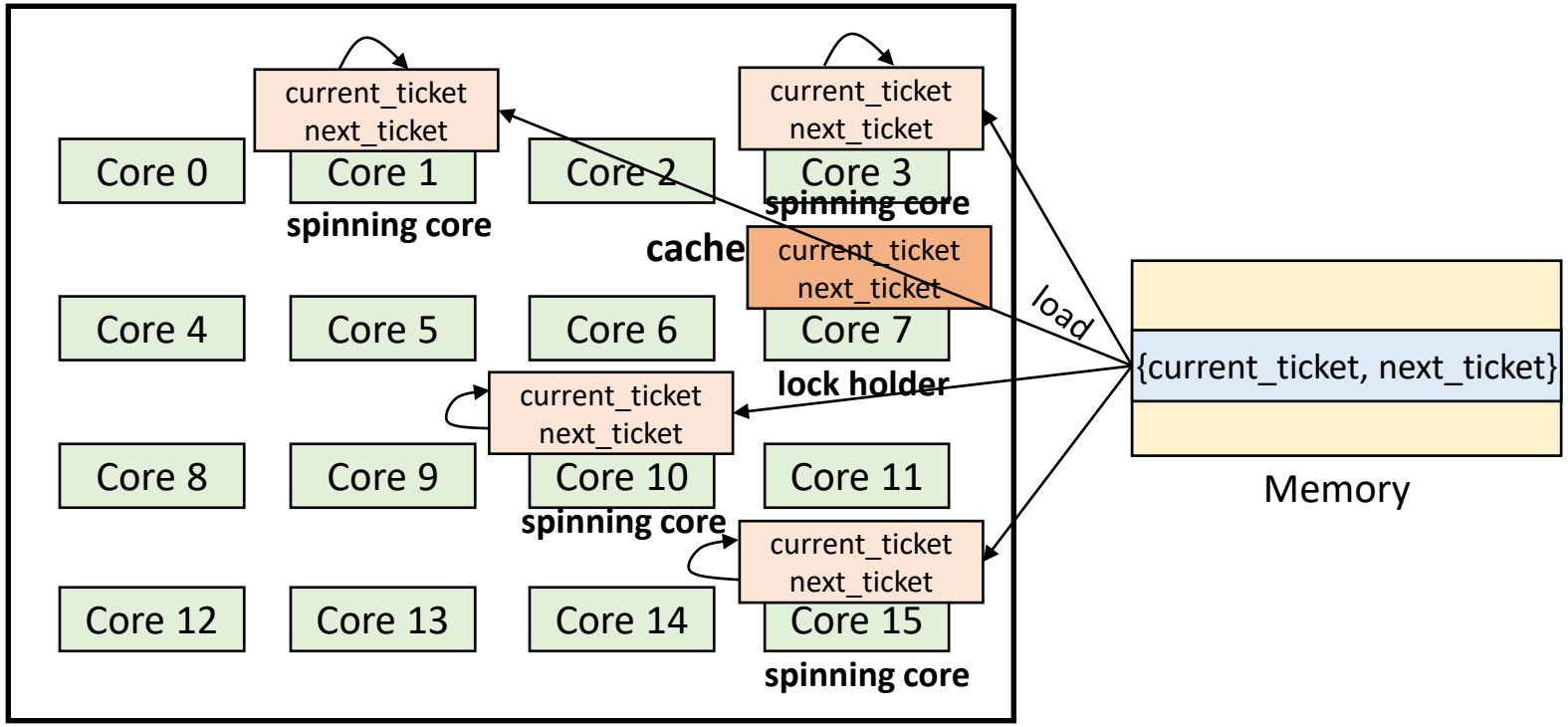
# Ticket spinlock: Acquire a spinlock

```
struct spinlock_t {
        int current_ticket;
        int next_ticket;
};

void spin_lock(spinlock_t *lock)
{
        int t = atomic_fetch_and_inc(&lock->next_ticket);
        while (t != lock ->current_ticket)
                ; /* spin */
}

void spin_unlock(spinlock_t *lock)
{
        lock ->current_ticket++;
}
```

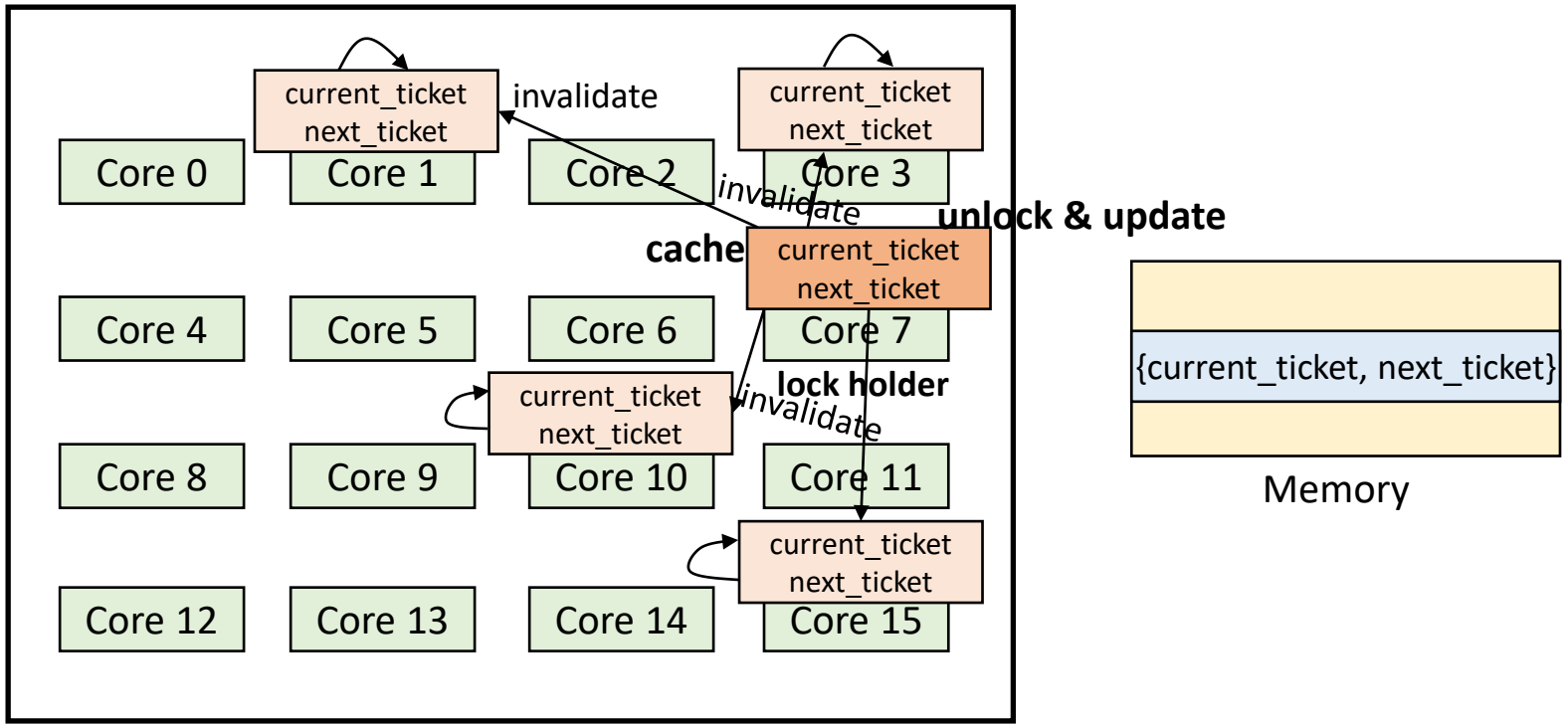


# Ticket spinlock: Other cores acquire a locked spinlock

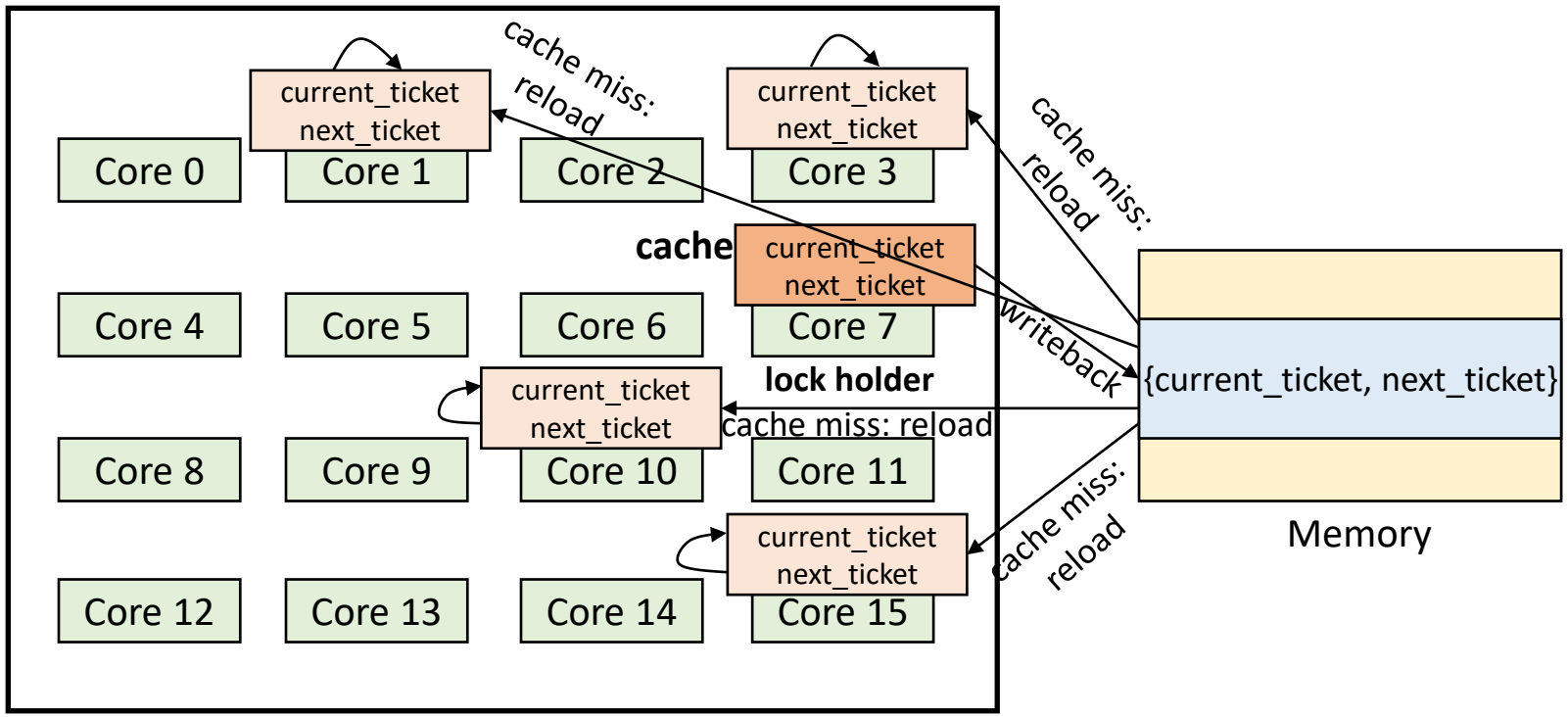
```
struct spinlock_t {
        int current_ticket;
        int next_ticket;
};

void spin_lock(spinlock_t *lock)
{
        int t = atomic_fetch_and_inc(&lock->next_ticket);
        while (t != lock ->current_ticket)
                ; /* spin */
}

void spin_unlock(spinlock_t *lock)
{
        lock ->current_ticket++;
}
```



# Ticket spinlock: Lock holder unlocks a spinlock and accumulates current_ticket

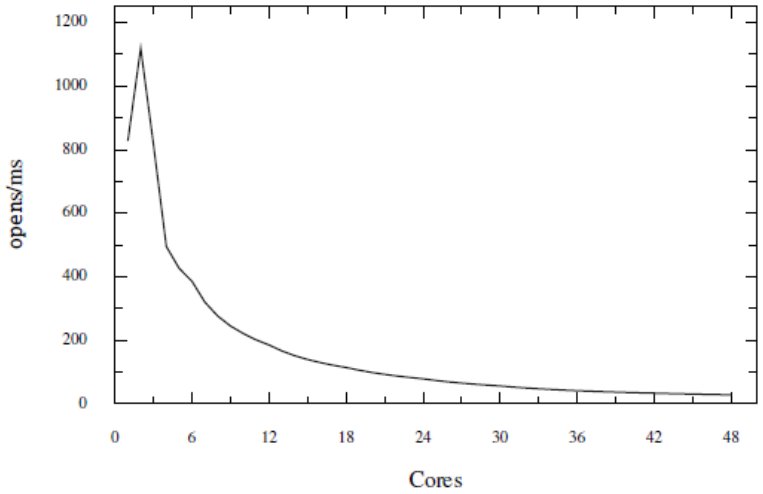


# Ticket spinlock: Cache miss → Reload

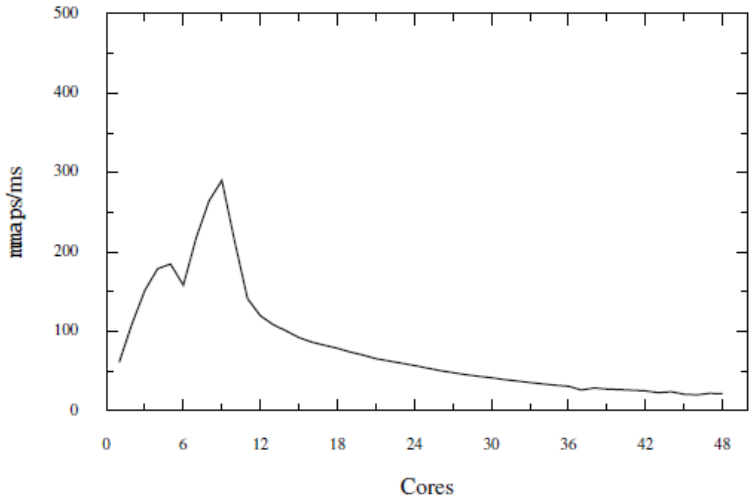



**[Ticket spinlock] Cache coherence issue: non-scalable spinlock (Cache-line bouncing)**

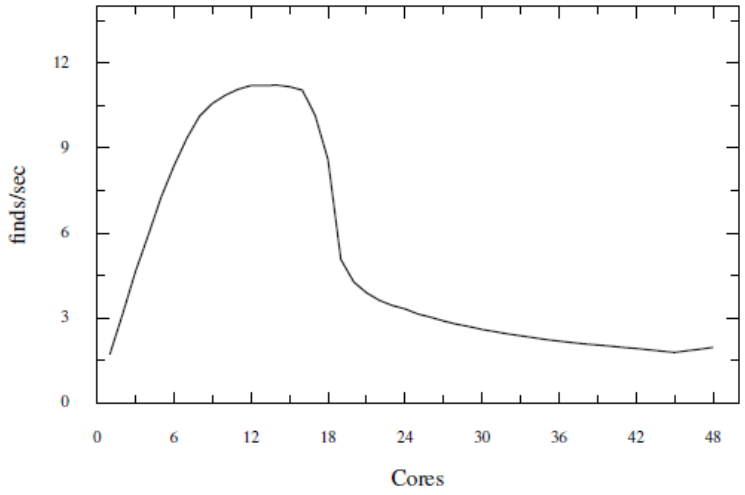
# Ticket spinlock: Performance Measurement



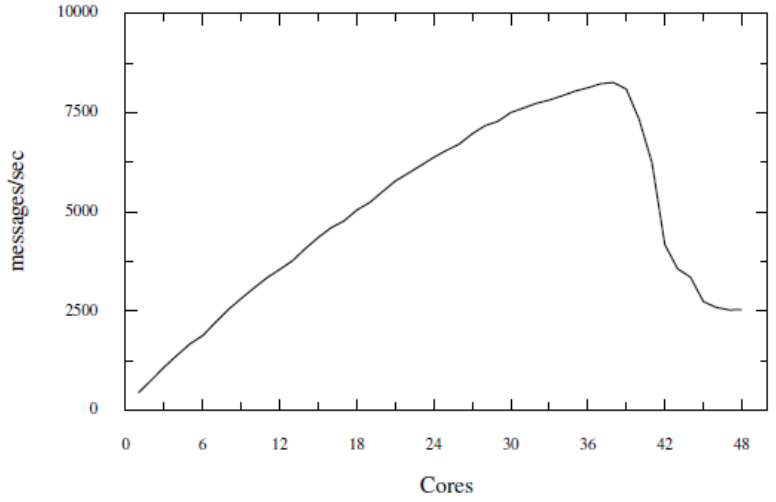

(a) Collapse for FOPS.




(b) Collapse for MEMPOP.




(c) Collapse for PFIND.




(d) Collapse for EXIM.

Figure 2: Sudden performance collapse with ticket locks.

**Benchmarks**

- FOPS: creates a single file and starts one process on each core. Repeated system calls “open() and close()”.
- MEMPOP
  - ✓ One process per core
  - ✓ Each process mmaps 64 kB of memory with the MAP_POPULATE flag, then munmaps the memory
- PFIND: searches for a file by executing several instances of the GNU find utility
- EXIM (mail server): A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection

**Reference Paper**: Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." Proceedings of the Linux Symposium. 2012.

# Ticket spinlock: Performance Measurement

| Benchmark | Operation time (cycles) | Top lock instance name | Acquires per operation | Average critical section time (cycles) | % of operation in critical section |
|---|---|---|---|---|---|
| FOPS | 503 | d_entry | 4 | 92 | 73% |
| MEMPOP | 6852 | anon_vma | 4 | 121 | 7% |
| PFIND | 2099 M | address_space | 70 K | 350 | 7% |
| EXIM | 1156 K | anon_vma | 58 | 165 | 0.8% |

Figure 3: The most contended critical sections for each Linux microbenchmark, on a single core.

**Reference Paper**: Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." Proceedings of the Linux Symposium. 2012.

# MCS (Mellor-Crummey & Scott) lock

```
struct mcs_spinlock {
        struct mcs_spinlock *next;
        int locked; /* 1 if lock acquired */
        int count;  /* nesting count, see qspinlock.c */
};
```

kernel/locking/mcs_spinlock.h

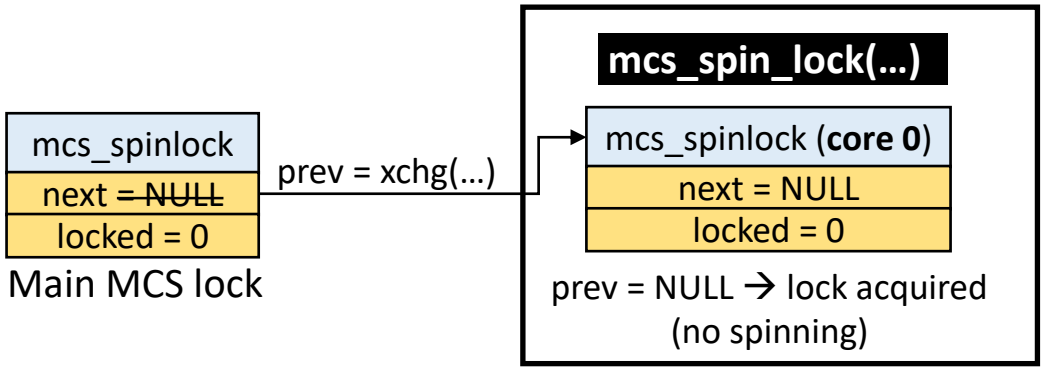
**MCS Lock**

- Adhere to fairness: FIFO (Implemented via linked list)
- Scalable spinlock: Prevent cache coherence issue
  - ✓ Each core reference its own data ‘next’ (struct mcs_spinlock)
    - Data ownership concept: Is Parallel Programming Hard, And, If So, What Can You Do About It?

# MCS (Mellor-Crummey & Scott) lock

```
struct mcs_spinlock {
        struct mcs_spinlock *next;
        int locked; /* 1 if lock acquired */
        int count;  /* nesting count, see qspinlock.c */
};
```

kernel/locking/mcs_spinlock.h




```
static inline
void mcs_spin_lock(struct mcs_spinlock **lock, struct mcs_spinlock *node)
{
        struct mcs_spinlock *prev;

        /* Init node */
        node->locked = 0;
        node->next   = NULL;

+--  6 lines: We rely on the full barrier with global transitivity implied by th
        prev = xchg(lock, node);
        if (likely(prev == NULL)) {
                /*
                 * Lock acquired, don't need to set node->locked to 1. Threads
                 * only spin on its own node->locked value for lock acquisition.
                 * However, since this thread can immediately acquire the lock
                 * and does not proceed to spin on its own node->locked, this
                 * value won't be used. If a debug mode is needed to
                 * audit lock status, then set node->locked value here.
                 */
                return;
        }
        WRITE_ONCE(prev->next, node);

        /* Wait until the lock holder passes the lock down. */
        arch_mcs_spin_lock_contended(&node->locked);
}
```

kernel/locking/mcs_spinlock.h 62,8 70%

1. **Adhere to fairness**
2. **Scalable spinlock: Prevent cache coherence issue**

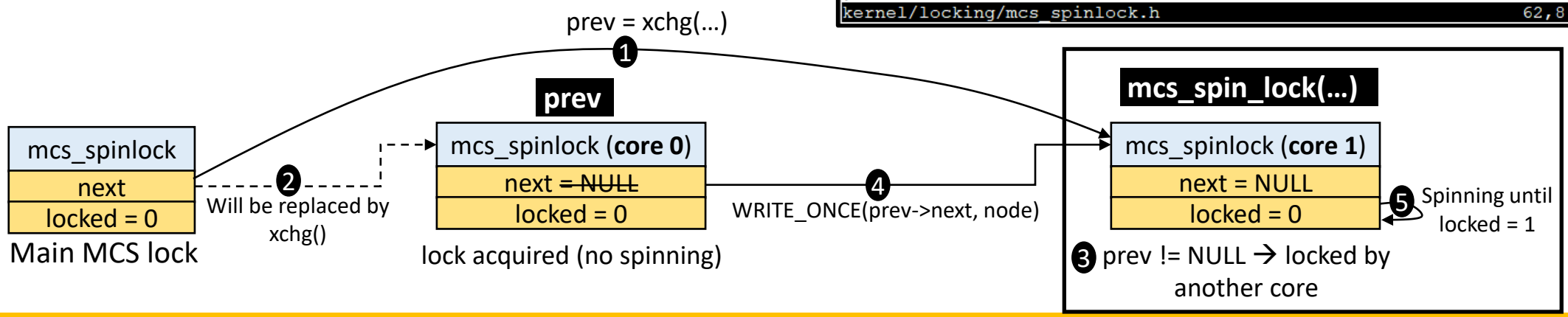
# MCS (Mellor-Crummey & Scott) lock

```
struct mcs_spinlock {
        struct mcs_spinlock *next;
        int locked; /* 1 if lock acquired */
        int count;  /* nesting count, see qspinlock.c */
};
```

kernel/locking/mcs_spinlock.h

```
static inline
void mcs_spin_lock(struct mcs_spinlock **lock, struct mcs_spinlock *node)
{
        struct mcs_spinlock *prev;

        /* Init node */
        node->locked = 0;
        node->next   = NULL;

+--  6 lines: We rely on the full barrier with global transitivity implied by th
        prev = xchg(lock, node);
        if (likely(prev == NULL)) {
                /*
                 * Lock acquired, don't need to set node->locked to 1. Threads
                 * only spin on its own node->locked value for lock acquisition.
                 * However, since this thread can immediately acquire the lock
                 * and does not proceed to spin on its own node->locked, this
                 * value won't be used. If a debug mode is needed to
                 * audit lock status, then set node->locked value here.
                 */
                return;
        }
        WRITE_ONCE(prev->next, node);

        /* Wait until the lock holder passes the lock down. */
        arch_mcs_spin_lock_contended(&node->locked);
}
```

kernel/locking/mcs_spinlock.h 62,8 70%

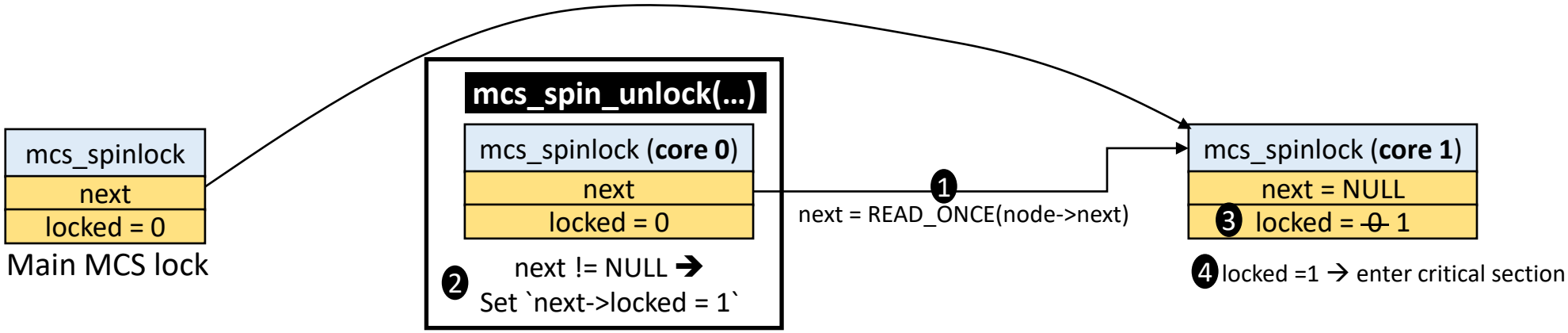



1. **[Spinning core] check its owner `locked`**
2. **'next' pointer of the main MCS lock indicates the tail of the queue of waiting cores**

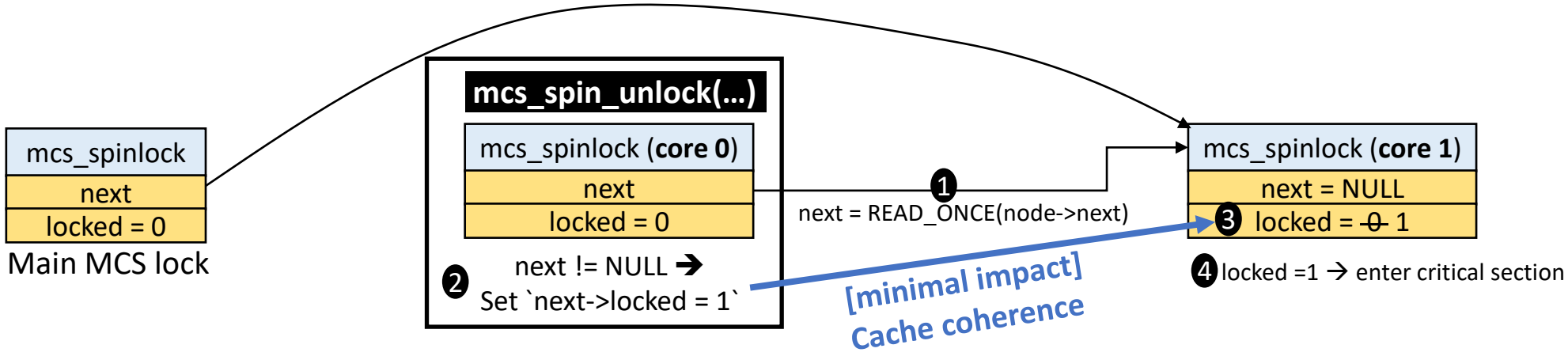
# MCS (Mellor-Crummey & Scott) lock

```
static inline
void mcs_spin_unlock(struct mcs_spinlock **lock, struct mcs_spinlock *node)
{
        struct mcs_spinlock *next = READ_ONCE(node->next);

        if (likely(!next)) {
                /*
                 * Release the lock by setting it to NULL
                 */
                if (likely(cmpxchg_release(lock, node, NULL) == node))
                        return;
                /* Wait until the next pointer is set */
                while (!(next = READ_ONCE(node->next)))
                        cpu_relax();
        }

        /* Pass lock to next waiter. */
        arch_mcs_spin_unlock_contended(&next->locked);
}
```

kernel/locking/mcs_spinlock.h 92,0-1



# MCS (Mellor-Crummey & Scott) lock




**Cache coherence only happens for the next core (will not be spinning; will enter critical section)**

# MCS (Mellor-Crummey & Scott) lock

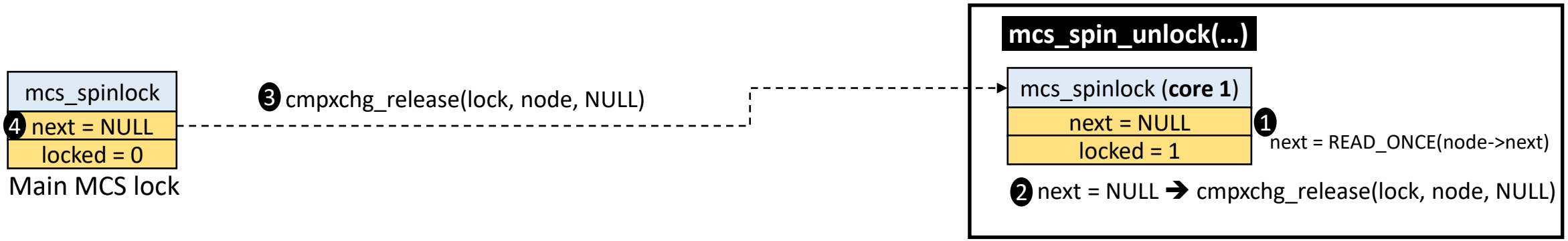
```
static inline
void mcs_spin_unlock(struct mcs_spinlock **lock, struct mcs_spinlock *node)
{
        struct mcs_spinlock *next = READ_ONCE(node->next);

        if (likely(!next)) {
                /*
                 * Release the lock by setting it to NULL
                 */
                if (likely(cmpxchg_release(lock, node, NULL) == node))
                        return;
                /* Wait until the next pointer is set */
                while (!(next = READ_ONCE(node->next)))
                        cpu_relax();
        }

        /* Pass lock to next waiter. */
        arch_mcs_spin_unlock_contended(&next->locked);
}
```

kernel/locking/mcs_spinlock.h 92,0-1

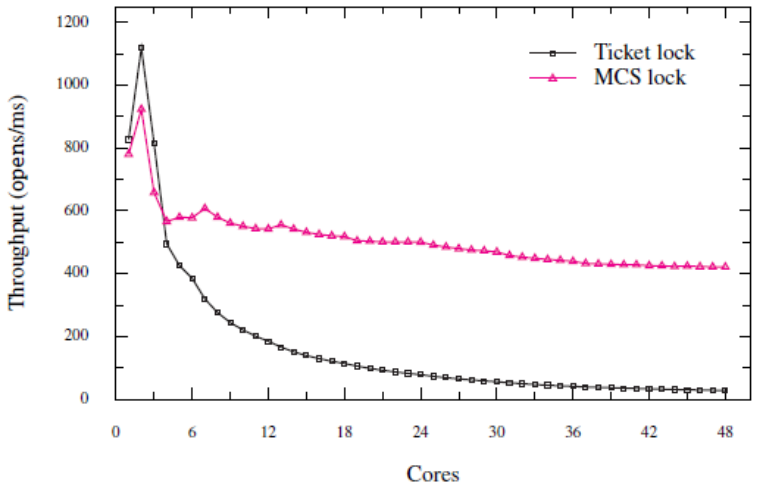
# MCS (Mellor-Crummey & Scott) lock

```
static inline
void mcs_spin_unlock(struct mcs_spinlock **lock, struct mcs_spinlock *node)
{
        struct mcs_spinlock *next = READ_ONCE(node->next);

        if (likely(!next)) {
                /*
                 * Release the lock by setting it to NULL
                 */
                if (likely(cmpxchg_release(lock, node, NULL) == node))
                        return;
                /* Wait until the next pointer is set */
                while (!(next = READ_ONCE(node->next)))
                        cpu_relax();
        }

        /* Pass lock to next waiter. */
        arch_mcs_spin_unlock_contended(&next->locked);
}
```

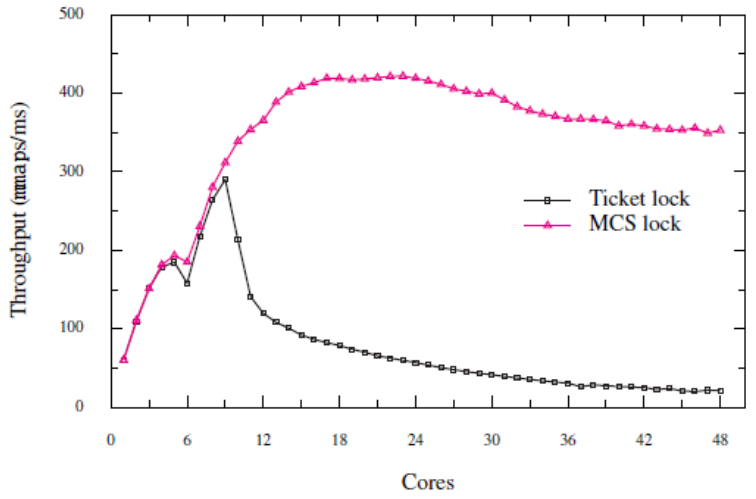
kernel/locking/mcs_spinlock.h 92,0-1



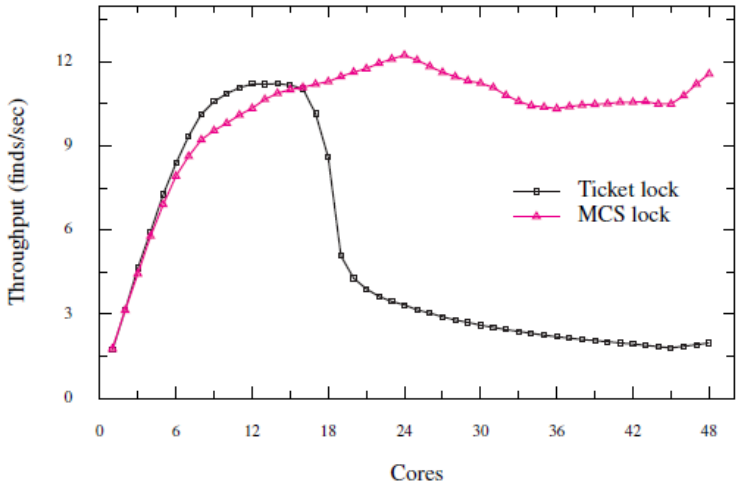
# Ticket spinlock vs MCS lock



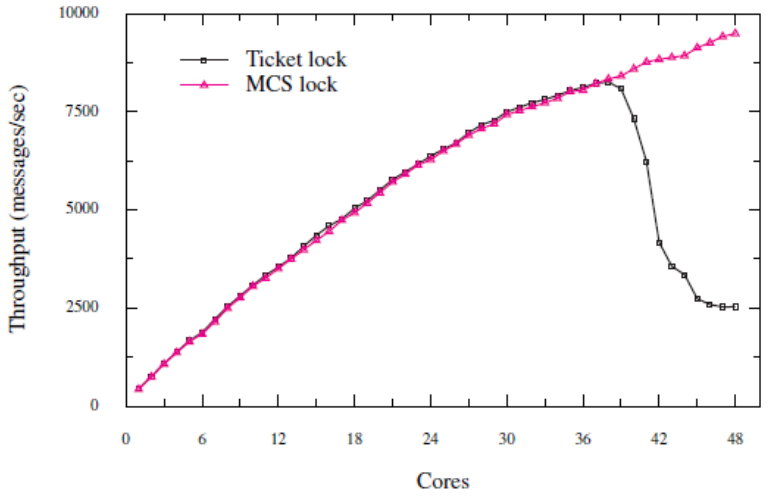

(a) Performance for FOPS.




(b) Performance for MEMPOP.




(c) Performance for PFIND.




(d) Performance for EXIM.

Figure 12: Performance of benchmarks using ticket locks and MCS locks.

## Benchmarks

- FOPS: creates a single file and starts one process on each core. Repeated system calls "open() and close()".
  - ✓ Executing the critical section increases from 450 cycles on two cores to 852 cycles on four cores
  - ✓ The critical section is executed multiple times per-operation and modifies shared data, which incurs costly cache misses
- MEMPOP
  - ✓ One process per core
  - ✓ Each process mmaps 64 kB of memory with the MAP_POPULATE flag, then munmaps the memory
- PFIND: searches for a file by executing several instances of the GNU find utility
- EXIM (mail server): A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection

**Reference Paper**: Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." Proceedings of the Linux Symposium. 2012.

# MCS Lock History in Linux Kernel

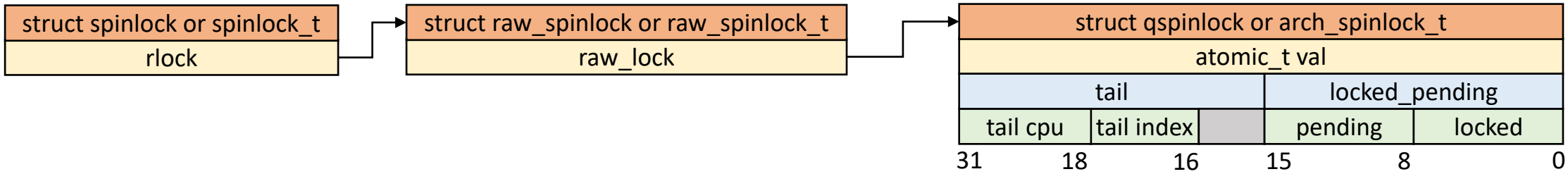
```
struct mcs_spinlock {
        struct mcs_spinlock *next;
        int locked; /* 1 if lock acquired */
        int count;  /* nesting count, see qspinlock.c */
};
```

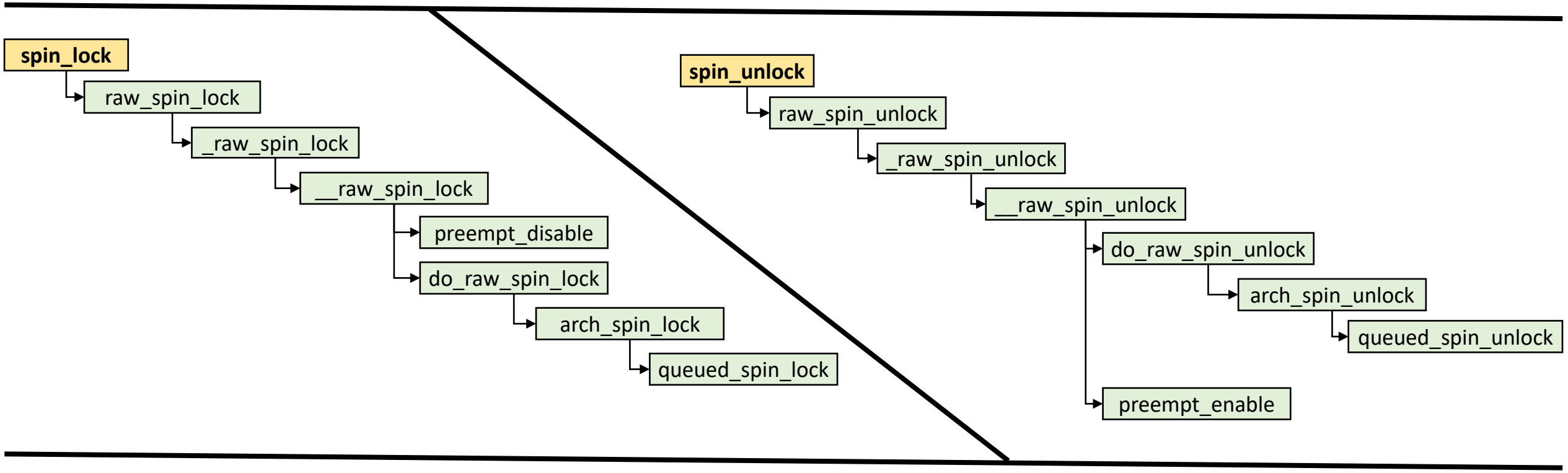
kernel/locking/mcs_spinlock.h

- Two variants
  - ✓ Standard: v3.15
    - ➢ None of Linux kernel subsystems uses it (No one calls mcs_spin_lock() and mcs_spin_unlock()) because:
      - sizeof(struct mcs_spinlock) > 4
        - ☐ Note: sizeof (msc spinlock struct) = 16
      - spinlock struct is embedded into kernel structure. Example:
        - ☐ struct page (size = 64) cannot tolerate the increased size
    - ➢ kernel/locking/mcs_spinlock.h
    - ➢ Replacement of ticket spinlock: qspinlock – based on standard MCS lock (v4.2)
      - A simple generic 4-byte queued spinlock
      - qspinlock is the default spinlock mechanism
  - ✓ Cancelable MCS lock (OSQ - Optimistic Spin Queue: MCS-like lock): v3.15
    - ➢ Used by mutex implementation & rw-semaphores
    - ➢ Mutex implementation paths:
      - Fastpath: Uncontended case by using cmpxchg()
      - Midpath (optimistic spinning) - The priority of the lock owner is the highest one
        - ☐ Spin for mutex lock acquisition when the lock owner is running.
        - ☐ The lock owner is likely to release the lock soon.
      - Slowpath: The task is added to the waiting queue and sleeps until woken up by the unlock path.
    - ➢ Mutex is a hybrid type (spinning & sleeping): Busy-waiting for a few cycles instead of immediately sleeping
    - ➢ kernel/locking/{mutex.c, osq_lock.c}
    - ➢ Reference: Generic Mutex Subsystem

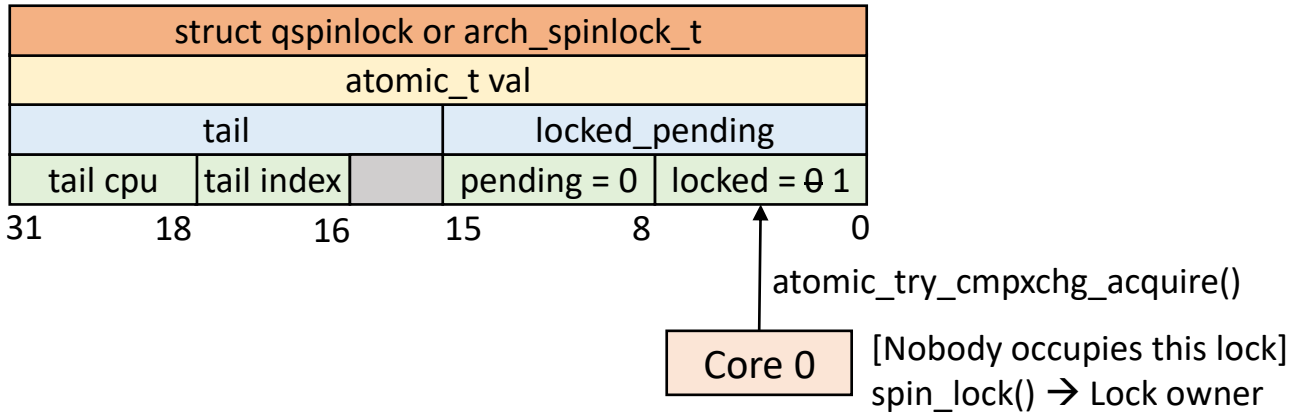
# qspinlock (Queue spinlock)

**spin_lock(), spin_unlock()**

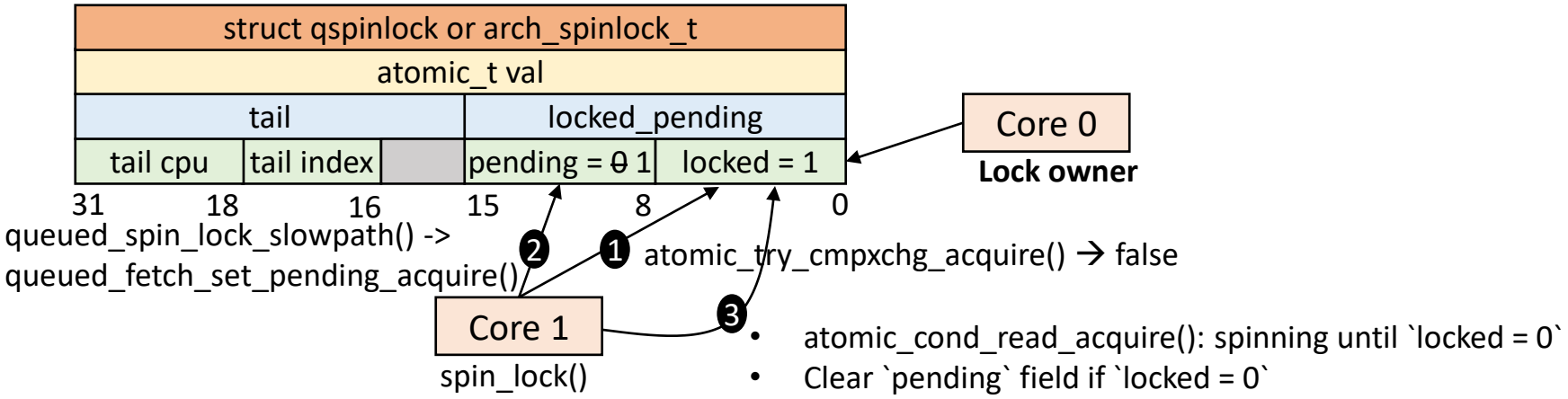


# qspinlock (Queue spinlock) – First core to acquire the lock

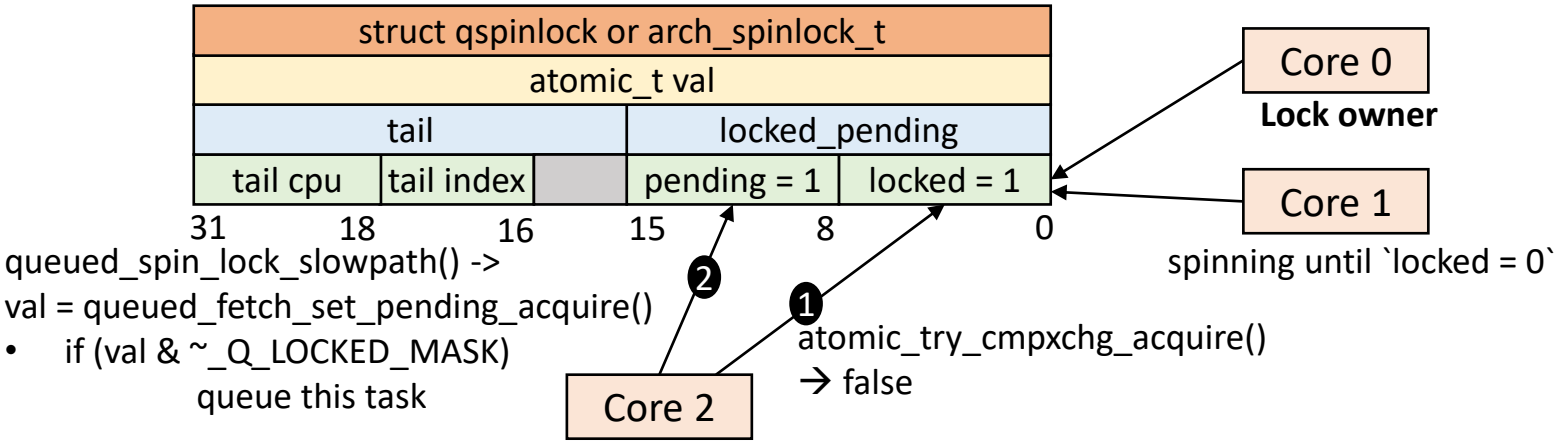


# qspinlock – Second core to acquire the lock

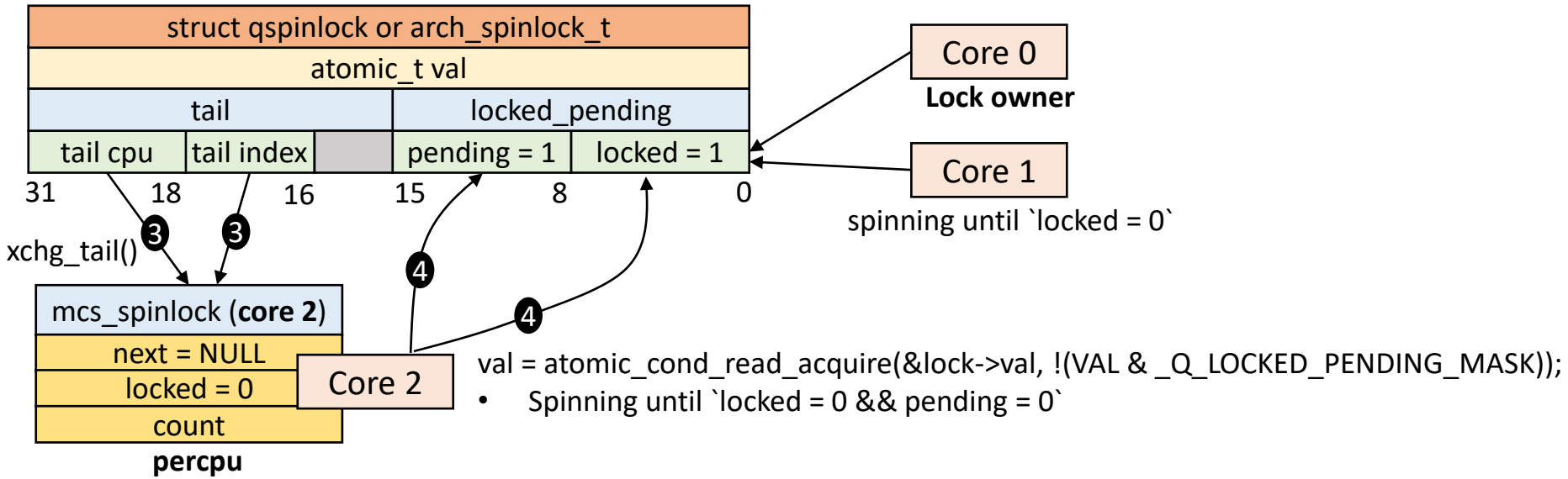



**[Second core for lock acquisition] Spinning by checking `locked` field of arch_spinlock_t (Optimization: No need to configure a per-cpu MSC struct)**

# qspinlock – Third core to acquire the lock

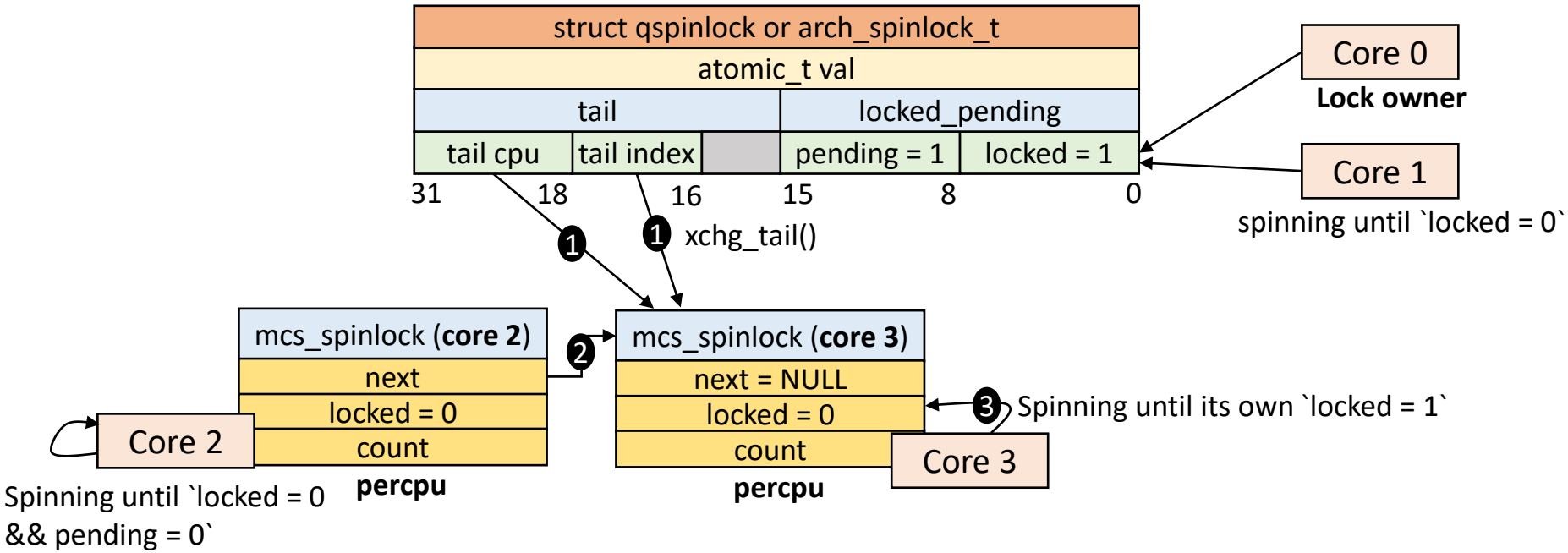


# qspinlock – Third core to acquire the lock




**[Third core for lock acquisition] Spinning until `locked = 0 && pending = 0`**

# qspinlock – Forth core to acquire the lock

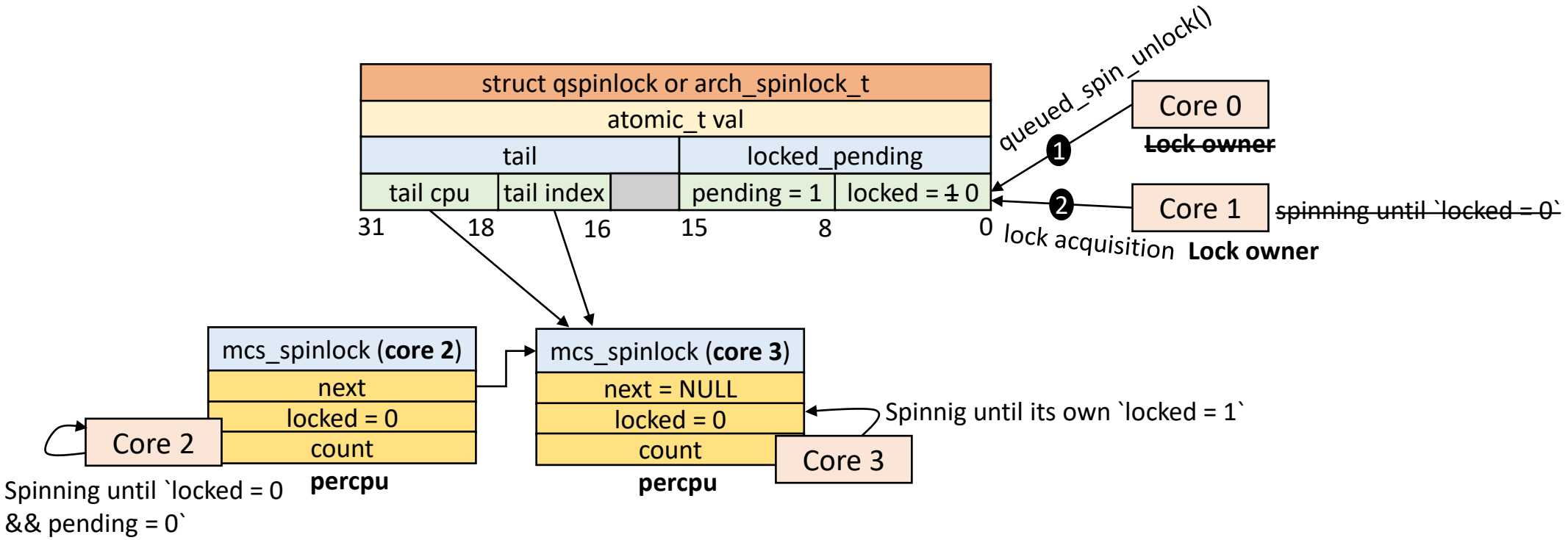



**[Forth and later core: N >= 4] Spinning until its own `locked = 1`: Improve cache bouncing**

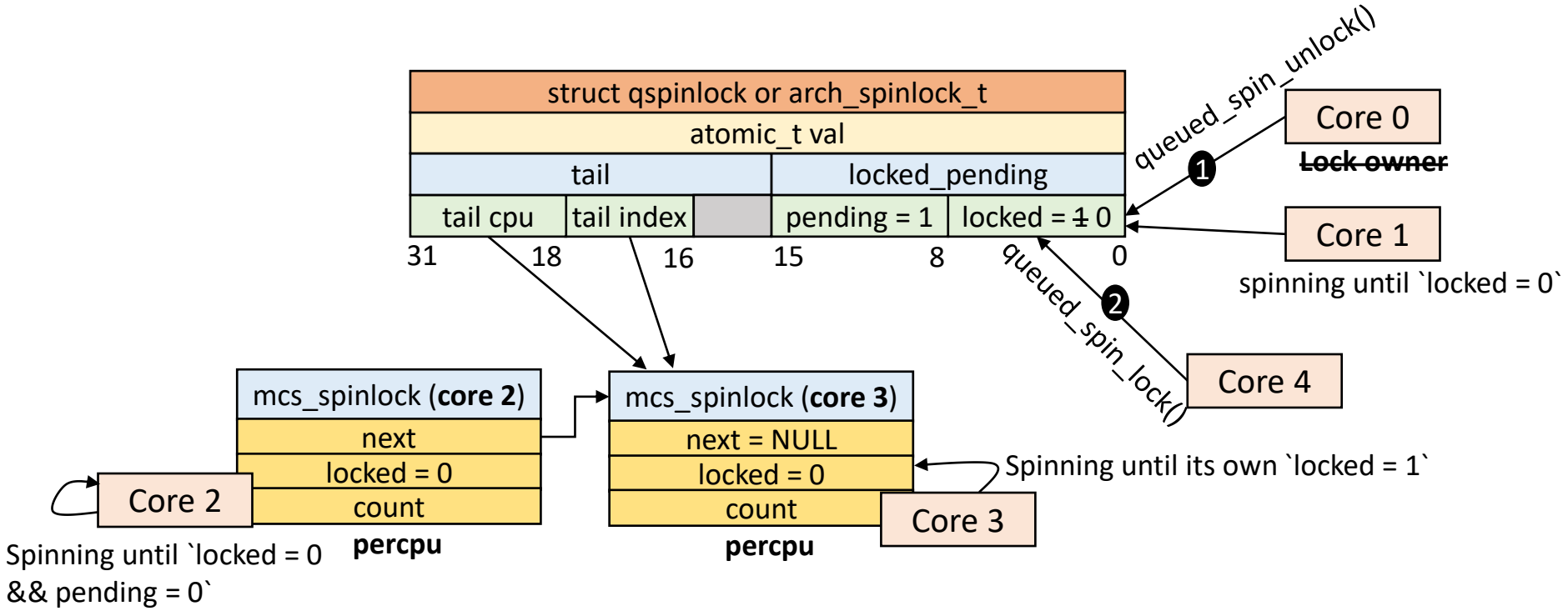
- **Who sets `mcs_spinlock.locked = 1`? (Hint: Adhere to MCS lock implementation)**

# qspinlock – queued_spin_unlock()

```
static __always_inline void queued_spin_unlock(struct qspinlock *lock)
{
        /*
         * unlock() needs release semantics:
         */
        smp_store_release(&lock->locked, 0);
}
include/asm-generic/qspinlock.h                                  76,3
```



# qspinlock – queued_spin_unlock() & queued_spin_lock()

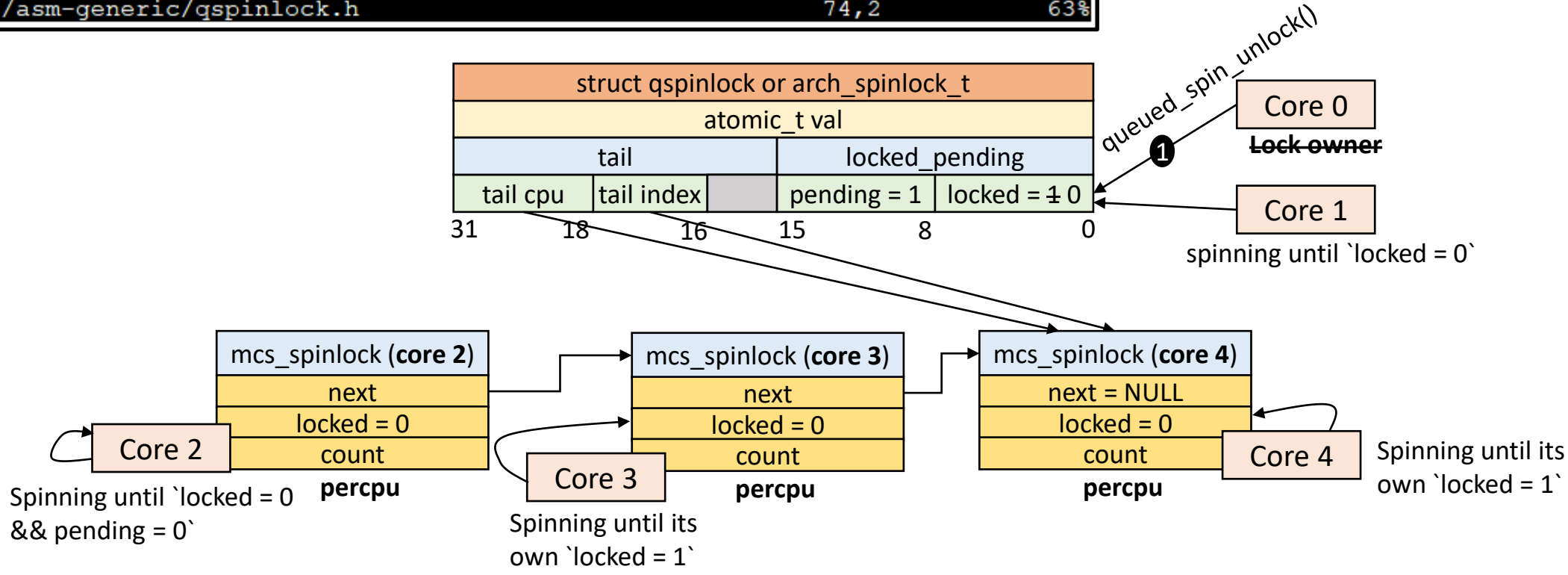



**What happens to core 4 for invoking queued_spin_lock() under this circumstance?**

# qspinlock – queued_spin_unlock() & queued_spin_lock()

```
static __always_inline void queued_spin_lock(struct qspinlock *lock)
{
        int val = 0;

        if (likely(atomic_try_cmpxchg_acquire(&lock->val, &val, _Q_LOCKED_VAL)))
                return;

        queued_spin_lock_slowpath(lock, val);
}
include/asm-generic/qspinlock.h                    74,2          63%
```

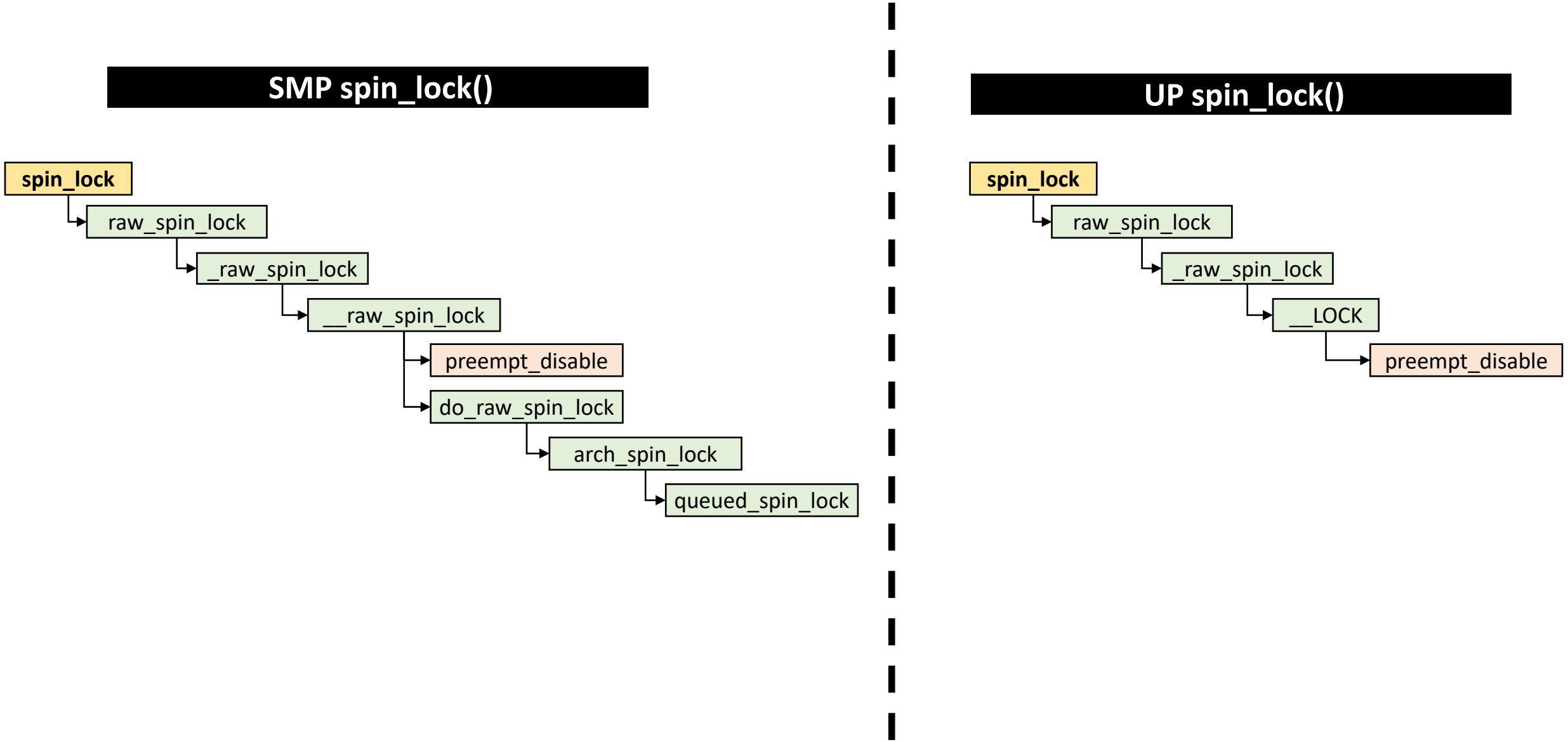
- atomic_try_cmpxchg_acquire(…)
  - Can acquire the lock (set 'locked=1) when the 32-bit qspinlock (val) is 0.
  - Otherwise, the core will go to the slowpath.




**1. Core 4 is queued: Adhere to the principle of ticket spinlock**

**2. Spin its own `mcs_spinlock.locked`: Improve cache bouncing**

# spin_lock() SMP & UP



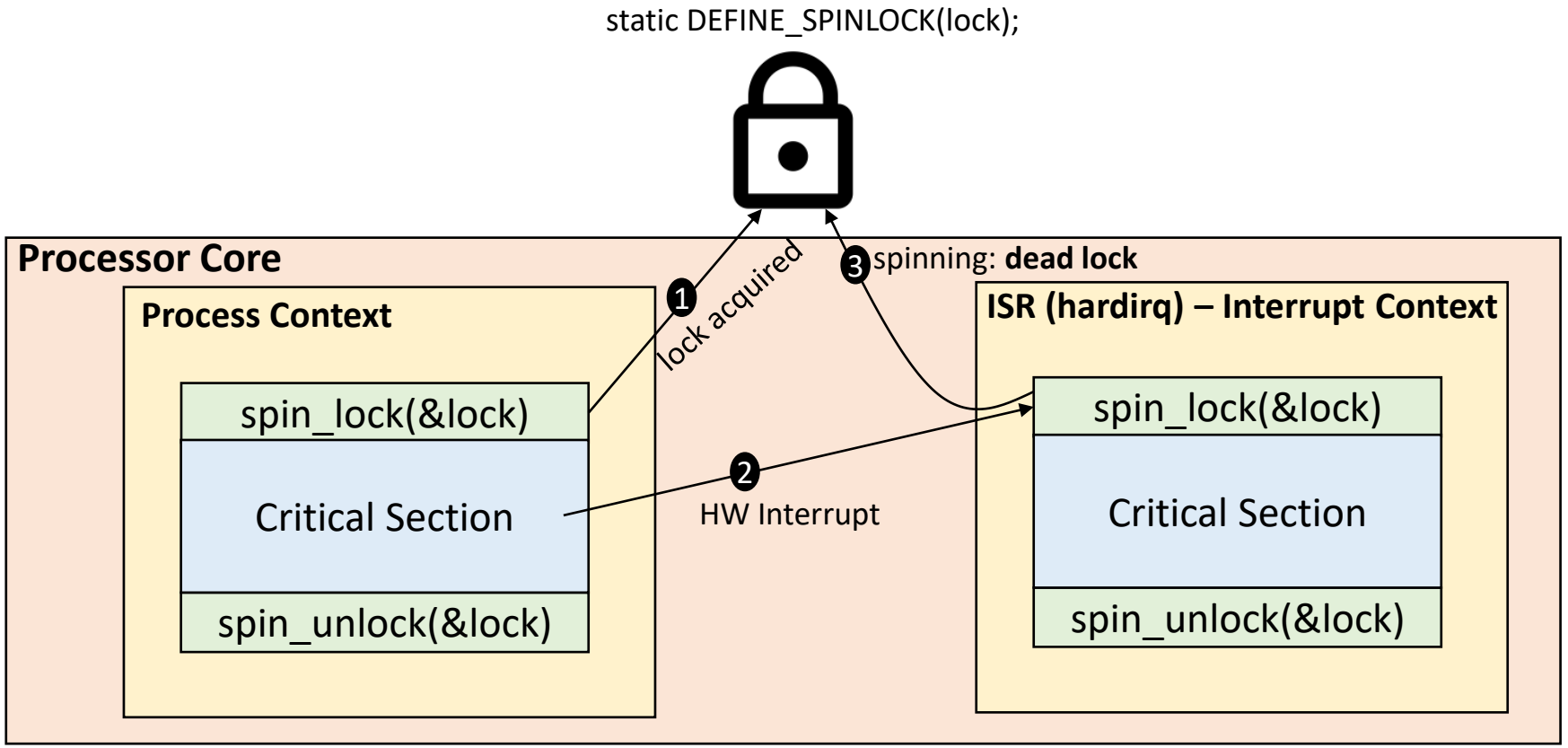
# spinlock() variants

- spin_lock() & spin_unlock()
  - ✓Multiple process contexts share the same data
- spin_lock_irq() & spin_unlock_irq()
  - ✓Process context and top halve (hardirq) share the same data
- spin_lock_irqsave() & raw_spin_unlock_irqrestore()
  - ✓Process context and top halve (hardirq) share the same data
  - ✓Save/restore eflags
- spin_lock_bh() & spin_unlock_bh()
  - ✓Process context and bottom halve share the same data
- spin_lock_nest_lock() & spin_lock_nested()
  - ✓lockdep: Annotate places where we take multiple locks of the same class and avoid deadlock – Commit b7d39aff9145 (“lockdep: spin_lock_nest_lock()”)

# spinlock() variants

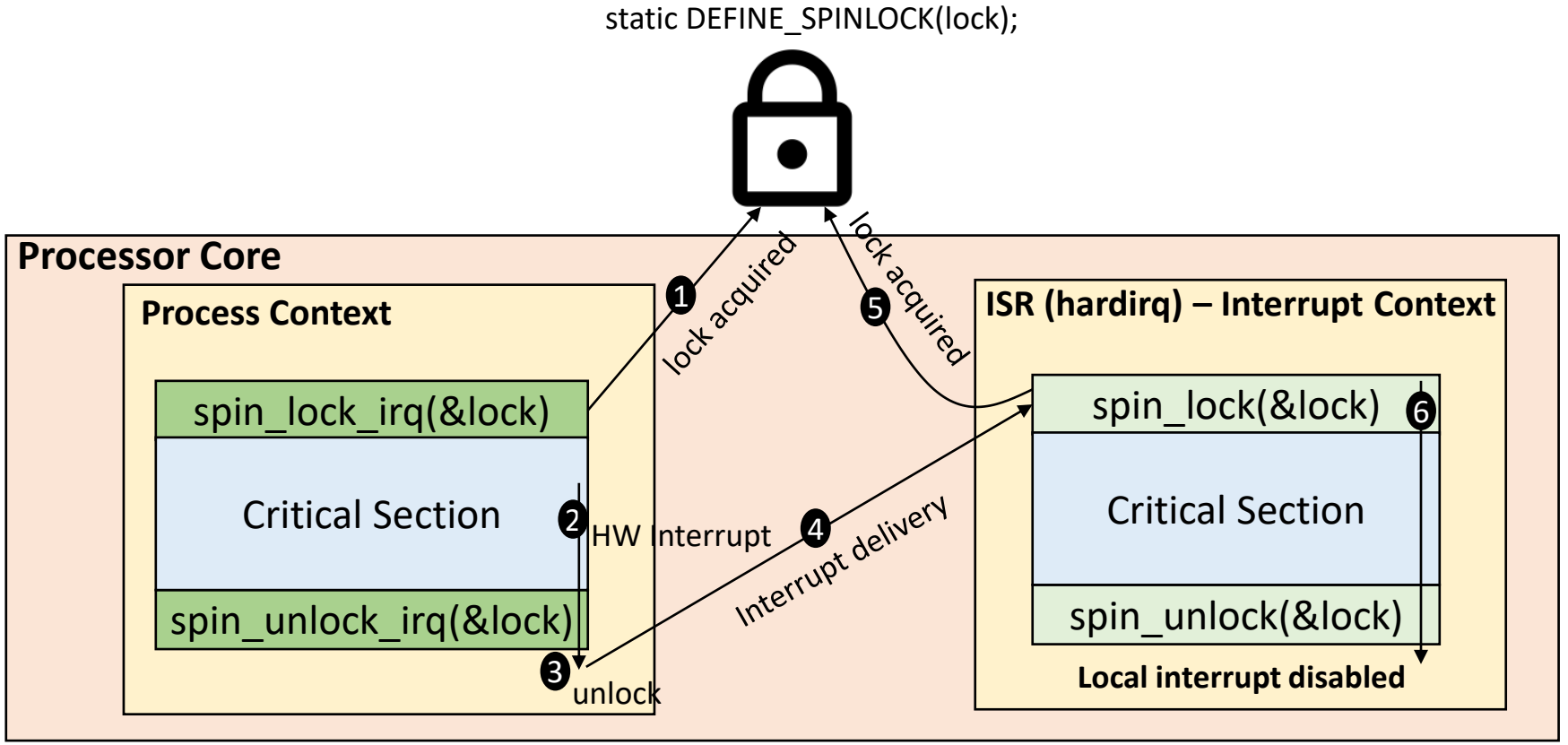
- spin_lock() & spin_unlock()
  - ✓Multiple process contexts share the same data
- spin_lock_irq() & spin_unlock_irq()
  - ✓Process context and top halve (hardirq) share the same data
- spin_lock_irqsave() & raw_spin_unlock_irqrestore()
  - ✓Process context and top halve (hardirq) share the same data
  - ✓Save/restore eflags
- spin_lock_bh() & spin_unlock_bh()
  - ✓Process context and bottom halve share the same data
- spin_lock_nest_lock() & spin_lock_nested()
  - ✓lockdep: Annotate places where we take multiple locks of the same class and avoid deadlock – Commit b7d39aff9145 (“lockdep: spin_lock_nest_lock()”)

# Deadlock between process context and interrupt context (the same core)



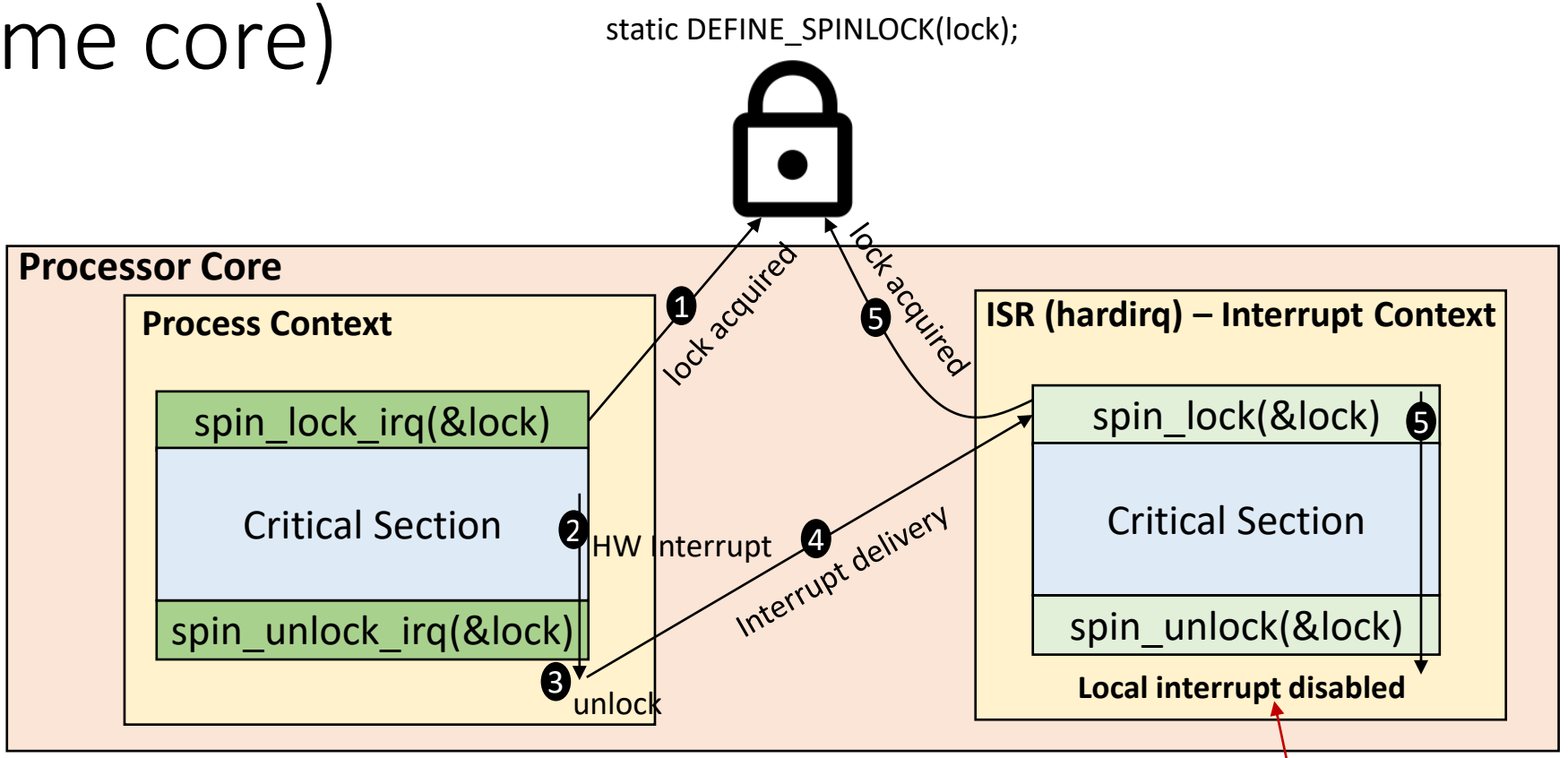

**Deadlock only happens on the same core in this circumstance**

# Deadlock between process context and interrupt context (the same core)




- **Use spin_{un}lock_irq() or spin_{un}lock_irqsave() to prevent deadlock**
  - ✓ **spin_lock_irq() or spin_lock_irqsave(): Disable local interrupt delivery**
  - ✓ **spin_unlock_irq() or spin_unlock_irqsave(): Enable local interrupt delivery**

# Deadlock between process context and interrupt context (the same core)




Who disables the local interrupt before entering ISR?

- Linux kernel common interrupt code?
- CPU HW?

# Does Linux kernel disable local interrupt before entering ISR?

## common_interrupt(): entry point for all normal device IRQs

```
/* Device interrupts common/spurious */
DECLARE_IDTENTRY_IRQ(X86_TRAP_OTHER,    common_interrupt);
#ifdef CONFIG_X86_LOCAL_APIC
DECLARE_IDTENTRY_IRQ(X86_TRAP_OTHER,    spurious_interrupt);
#endif
```

arch/x86/include/asm/idtentry.h

```
/* Entries for common/spurious (device) interrupts */
#define DECLARE_IDTENTRY_IRQ(vector, func)
        idtentry_irq vector func
```

arch/x86/include/asm/idtentry.h

```
.macro idtentry_irq vector cfunc
        .p2align CONFIG_X86_L1_CACHE_SHIFT
        idtentry \vector asm_\cfunc \cfunc has_error_code=1
.endm
```

arch/x86/entry/entry_64.S

```
.macro idtentry vector asmsym cfunc has_error_code:req
SYM_CODE_START(\asmsym)
        UNWIND_HINT_IRET_REGS offset=\has_error_code*8
        ASM_CLAC

        .if \has_error_code == 0
                pushq   $-1                             /* ORIG
rt */
        .endif

        .if \vector == X86_TRAP_BP
                /*
                 * If coming from kernel space, create
                 * int3 handler to emulate a call insti
                 */
                testb   $3, CS-ORIG_RAX(%rsp)
                jnz     .Lfrom_usermode_no_gap_\@
                .rept   6
                pushq   5*8(%rsp)
                .endr
                UNWIND_HINT_IRET_REGS offset=8
.Lfrom_usermode_no_gap_\@:
        .endif

        idtentry_body \cfunc \has_error_code

_ASM_NOKPROBE(\asmsym)
SYM_CODE_END(\asmsym)
.endm
```

SYM_CODE_START(asm_common_interrupt)

arch/x86/entry/entry_64.S

# Does Linux kernel disable local interrupt before entering ISR?

```
.macro idtentry vector asmsym cfunc has_error_code:req
SYM_CODE_START(\asmsym) → SYM_CODE_START(asm_common_interrupt)
        UNWIND_HINT_IRET_REGS offset=\has_error_code*8
        ASM_CLAC

        .if \has_error_code == 0
                pushq   $-1                             /* ORI
rt */
        .endif

        .if \vector == X86_TRAP_BP
                /*
                 * If coming from kernel space, create
                 * int3 handler to emulate a call inst
                 */
                testb   $3, CS-ORIG_RAX(%rsp)
                jnz     .Lfrom_usermode_no_gap_\@
                .rept   6
                pushq   5*8(%rsp)
                .endr
                UNWIND_HINT_IRET_REGS offset=8
.Lfrom_usermode_no_gap_\@:
        .endif

        idtentry_body \cfunc \has_error_code

_ASM_NOKPROBE(\asmsym)
SYM_CODE_END(\asmsym)
.endm

arch/x86/entry/entry_64.S
```

```
.macro idtentry_body cfunc has_error_code:req

        call    error_entry
        UNWIND_HINT_REGS

        movq    %rsp, %rdi                      /* pt_regs pointer into 1st argu
ment*/

        .if \has_error_code == 1
                movq    ORIG_RAX(%rsp), %rsi    /* get error code into 2nd argum
ent*/
                movq    $-1, ORIG_RAX(%rsp)     /* no syscall to restart */
        .endif

        call    \cfunc

        jmp     error_return
.endm

/**
arch/x86/entry/entry_64.S                       320,18          21%
```

```
$ objdump -D out/obj/linux/vmlinux
...
ffffffff81400b00 <asm_common_interrupt>:
ffffffff81400b00:       e8 fb 03 00 00          callq  ffffffff81400f00 <error_entry>
ffffffff81400b05:       48 89 e7                mov    %rsp,%rdi
ffffffff81400b08:       48 8b 74 24 78          mov    0x78(%rsp),%rsi
ffffffff81400b0d:       48 c7 44 24 78 ff ff    movq   $0xffffffffffffffff,0x78(%rsp)
ffffffff81400b14:       ff ff
ffffffff81400b16:       e8 65 b0 f6 ff          callq  ffffffff8136bb80 <common_interrupt>
ffffffff81400b1b:       e9 d0 04 00 00          jmpq   ffffffff81400ff0 <error_return>
ffffffff81400b20:       66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
ffffffff81400b27:       00 00 00 00
ffffffff81400b2b:       66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
ffffffff81400b32:       00 00 00 00
ffffffff81400b36:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
ffffffff81400b3d:       00 00 00
```

# Does Linux kernel disable local interrupt before entering ISR?

```
$ objdump -D out/obj/linux/vmlinux
...
ffffffff81400b00 <asm_common_interrupt>:
ffffffff81400b00:       e8 fb 03 00 00          callq  ffffffff81400f00 <error_entry>
ffffffff81400b05:       48 89 e7                mov    %rsp,%rdi
ffffffff81400b08:       48 8b 74 24 78          mov    0x78(%rsp),%rsi
ffffffff81400b0d:       48 c7 44 24 78 ff ff    movq   $0xffffffffffffffff,0x78(%rsp)
ffffffff81400b14:       ff ff
ffffffff81400b16:       e8 65 b0 f6 ff          callq  ffffffff8136bb80 <common_interrupt>
ffffffff81400b1b:       e9 d0 04 00 00          jmpq   ffffffff81400ff0 <error_return>
ffffffff81400b20:       66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
ffffffff81400b27:       00 00 00 00
ffffffff81400b2b:       66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
ffffffff81400b32:       00 00 00 00
ffffffff81400b36:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
ffffffff81400b3d:       00 00 00
```

```
(gdb) bt 3
#0  0xffffffff81400b00 in asm_common_interrupt ()
#1  0x0000000000000030 in fixed_percpu_data ()
#2  0xffffffff813764d5 in native_restore_fl (flags=582) at /home/adrian/git-repo
/gdb-linux-real-mode/src/linux-5.11/arch/x86/include/asm/irqflags.h:84
(More stack frames follow...)
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0xffffffff81400b00 <asm_common_interrupt>
        breakpoint already hit 1 time
2       breakpoint     keep y   0xffffffff8136bb80 in common_interrupt at /home/
adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/irq.c:239
(gdb) info registers eflags
eflags         0x46             [ IOPL=0 ZF PF ]
```

The current stack is not switched yet: will be done in error_entry()

← IF (Interrupt Enable Flag) is disabled!

**Local interrupt has been disabled before entering asm_common_interrupt(): CPU disables it.**

# Does Linux kernel disable local interrupt before entering ISR?

```
$ objdump -D out/obj/linux/vmlinux
...
ffffffff81400b00 <asm_common_interrupt>:
ffffffff81400b00:       e8 fb 03 00 00          callq  ffffffff81400f00 <error_entry>
ffffffff81400b05:       48 89 e7                mov    %rsp,%rdi
ffffffff81400b08:       48 8b 74 24 78          mov    0x78(%rsp),%rsi
ffffffff81400b0d:       48 c7 44 24 78 ff ff    movq   $0xffffffffffffffff,0x78(%rsp)
ffffffff81400b14:       ff ff
ffffffff81400b16:       e8 65 b0 f6 ff          callq  ffffffff8136bb80 <common_interrupt>
ffffffff81400b1b:       e9 d0 04 00 00          jmpq   ffffffff81400ff0 <error_return>
ffffffff81400b20:       66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
ffffffff81400b27:       00 00 00 00
ffffffff81400b2b:       66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
ffffffff81400b32:       00 00 00 00
ffffffff81400b36:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
ffffffff81400b3d:       00 00 00
```

```
Thread 1 hit Breakpoint 2, common_interrupt (regs=0xffffffff81a03d88, error_code
=48) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel
/irq.c:239
239     DEFINE_IDTENTRY_IRQ(common_interrupt)
(gdb) bt
#0  common_interrupt (regs=0xffffffff81a03d88, error_code=48) at /home/adrian/gi
t-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/irq.c:239
#1  0xffffffff81400b1b in asm_common_interrupt ()
#2  0x0000000000000000 in ?? ()
```

# Does Linux kernel disable local interrupt before entering ISR?

```
DEFINE_IDTENTRY_IRQ(common_interrupt)
{
        struct pt_regs *old_regs = set_irq_regs(regs);
        struct irq_desc *desc;
```

arch/x86/kernel/irq.c

```
#define DEFINE_IDTENTRY_IRQ(func)                                       \
static __always_inline void __##func(struct pt_regs *regs, u8 vector);  \
                                                                        \
__visible noinstr void func(struct pt_regs *regs,                       \
                            unsigned long error_code)                   \
{                                                                       \
        irqentry_state_t state = irqentry_enter(regs);                  \
                                                                        \
        instrumentation_begin();                                        \
        irq_enter_rcu();                                                \
        kvm_set_cpu_l1tf_flush_l1d();                                   \
        __##func (regs, (u8)error_code);                                \
        irq_exit_rcu();                                                 \
        instrumentation_end();                                          \
        irqentry_exit(regs, state);                                     \
}                                                                       \
                                                                        \
static __always_inline void __##func(struct pt_regs *regs, u8 vector)
```

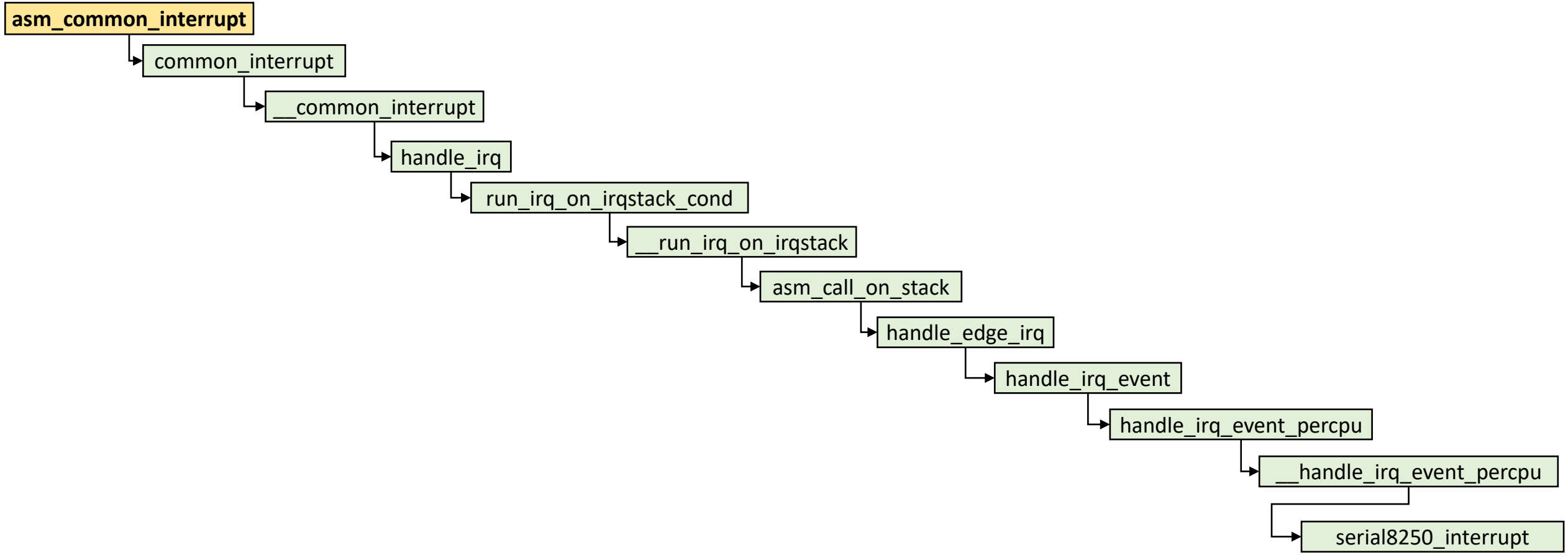
common_interrupt(…)

__common_interrupt(…)

arch/x86/include/asm/idtentry.h 182,1

# Does Linux kernel disable local interrupt before entering ISR?

## Common Interrupt Call Path: Serial Driver




**Common interrupt in Linux kernel: Handle vector number 32-255**

# Common Interrupt Code Path

```
(gdb) bt
#0  serial8250_interrupt (irq=4, dev_id=0xffff888240d52780) at /home/adrian/git-
repo/gdb-linux-real-mode/src/linux-5.11/include/linux/spinlock.h:354
#1  0xffffffff810822e0 in __handle_irq_event_percpu (desc=desc@entry=0xffff88810
004fa00, flags=flags@entry=0xffffc90000003f9c) at /home/adrian/git-repo/gdb-linu
x-real-mode/src/linux-5.11/kernel/irq/handle.c:156
#2  0xffffffff8108236f in handle_irq_event_percpu (desc=desc@entry=0xffff8881000
4fa00) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/irq/ha
ndle.c:196
#3  0xffffffff810823d7 in handle_irq_event (desc=desc@entry=0xffff88810004fa00)
at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/irq/handle.c:
213
#4  0xffffffff81085ee5 in handle_edge_irq (desc=0xffff88810004fa00) at /home/adr
ian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/irq/chip.c:819
#5  0xffffffff81400ddf in asm_call_on_stack () at /home/adrian/git-repo/gdb-linu
x-real-mode/src/linux-5.11/drivers/tty/serial/8250/8250_core.c:842
#6  0xffffffff8136bc38 in __run_irq_on_irqstack (desc=0xffff88810004fa00, func=<
optimized out>) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch
/x86/include/asm/irq_stack.h:48
#7  run_irq_on_irqstack_cond (regs=0xffffc900018a3a78, desc=0xffff88810004fa00,
func=<optimized out>) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.1
1/arch/x86/include/asm/irq_stack.h:101
#8  handle_irq (regs=0xffffc900018a3a78, desc=0xffff88810004fa00) at /home/adria
n/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/irq.c:230
#9  __common_interrupt (vector=36 '$', regs=0xffffc900018a3a78) at /home/adrian/
git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/irq.c:249
#10 common_interrupt (regs=0xffffc900018a3a78, error_code=36) at /home/adrian/gi
t-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/irq.c:239
#11 0xffffffff81400b1b in asm_common_interrupt () at /home/adrian/git-repo/gdb-l
inux-real-mode/src/linux-5.11/drivers/tty/serial/8250/8250_core.c:842
#12 0x0000000000000000 in ?? ()
```

# CPU disables local interrupt if the interrupt handle is called via an interrupt gate

### 6.5.1 Call and Return Operation for Interrupt or Exception Handling Procedures

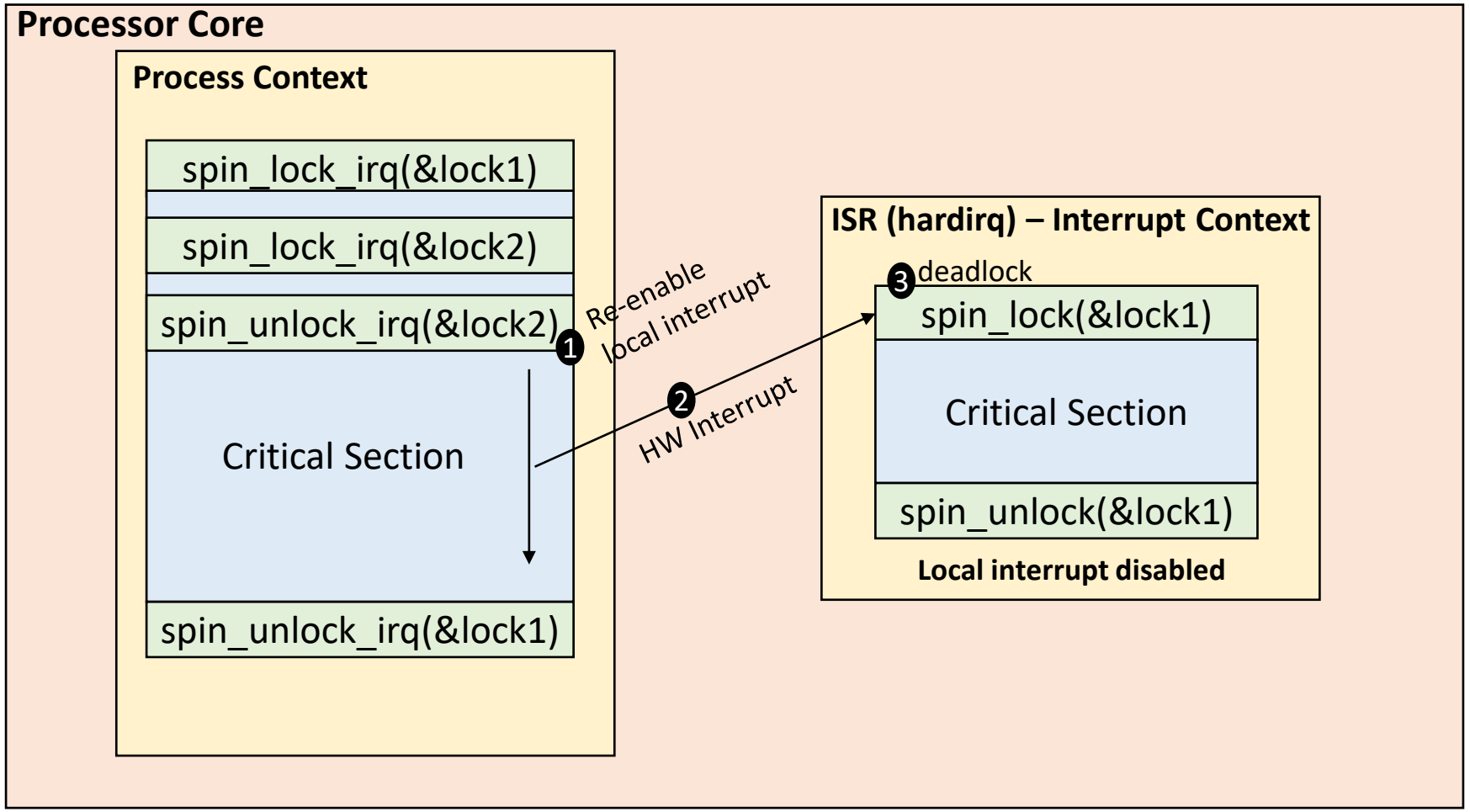
A call to an interrupt or exception handler procedure is similar to a procedure call to another protection level (see Section 6.4.6, "CALL and RET Operation Between Privilege Levels"). Here, the vector references one of two kinds of gates in the IDT: an **interrupt gate** or a **trap gate**. Interrupt and trap gates are similar to call gates in that they provide the following information:

- Access rights information
- The segment selector for the code segment that contains the handler procedure
- An offset into the code segment to the first instruction of the handler procedure

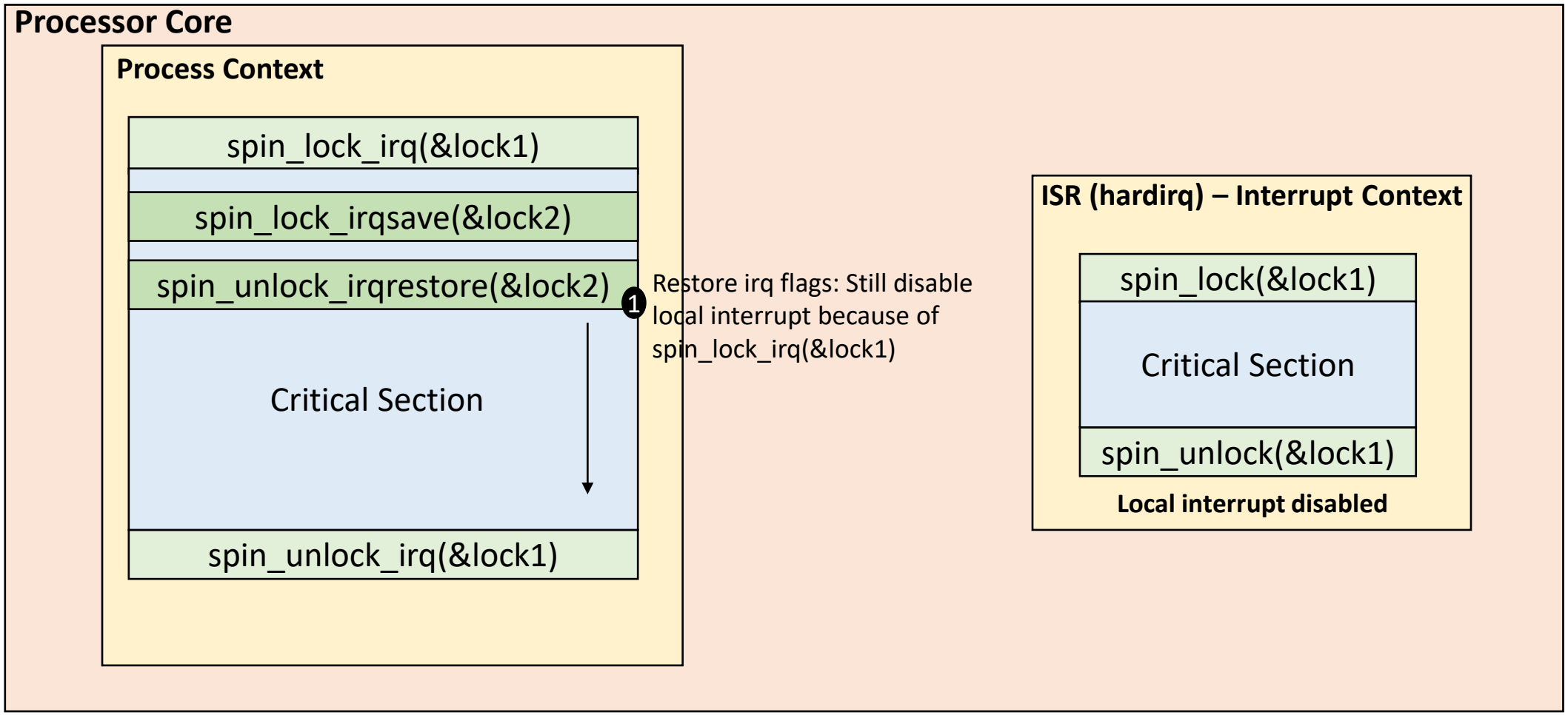
The difference between an interrupt gate and a trap gate is as follows. If an interrupt or exception handler is called through an interrupt gate, the processor clears the interrupt enable (IF) flag in the EFLAGS register to prevent subsequent interrupts from interfering with the execution of the handler. When a handler is called through a trap gate, the state of the IF flag is not changed.

[Reference] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

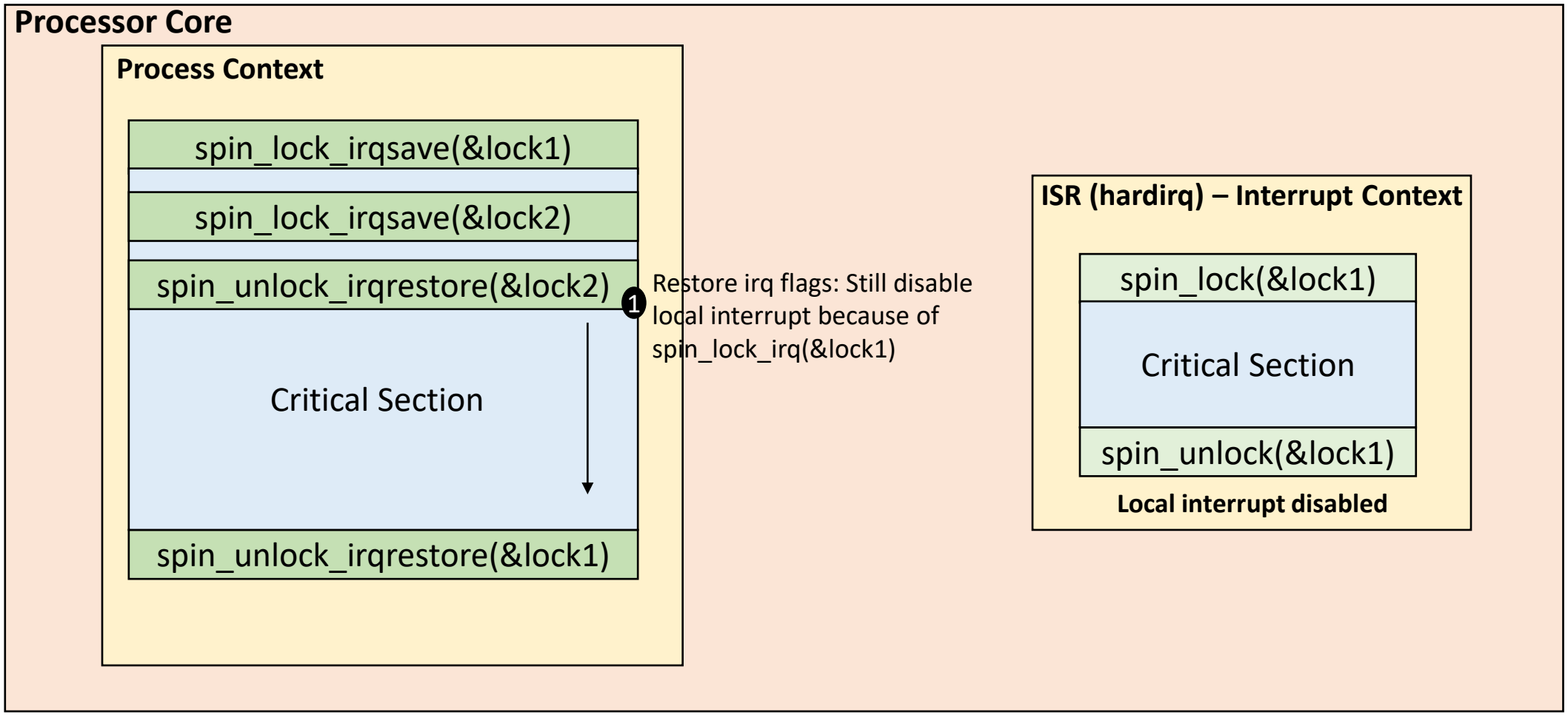
# Deadlock between process context and interrupt context (the same core)



# Deadlock between process context and interrupt context (the same core) – Solution 1



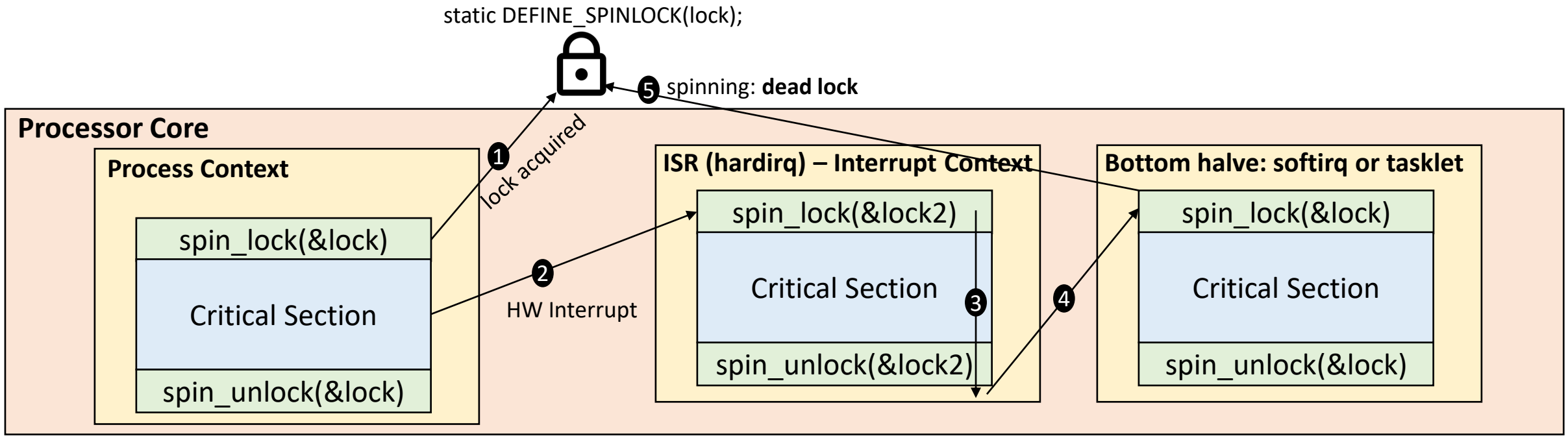
# Deadlock between process context and interrupt context (the same core) – Solution 2



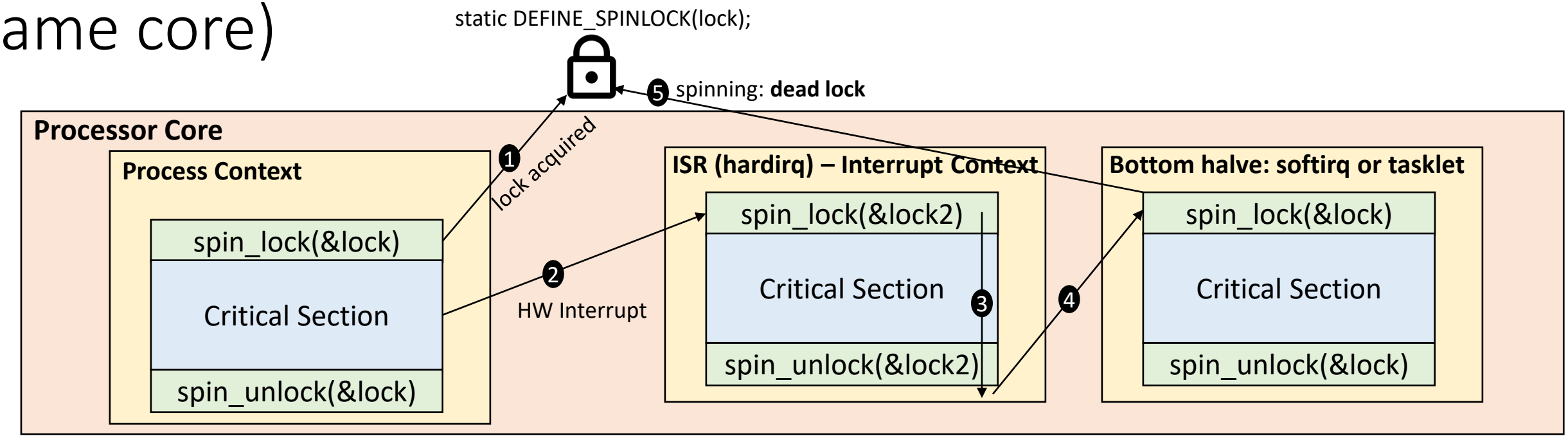
# spinlock() variants

- spin_lock() & spin_unlock()
  - ✓Multiple process contexts share the same data
- spin_lock_irq() & spin_unlock_irq()
  - ✓Process context and top halve (hardirq) share the same data
- spin_lock_irqsave() & raw_spin_unlock_irqrestore()
  - ✓Process context and top halve (hardirq) share the same data
  - ✓Save/restore eflags
- spin_lock_bh() & spin_unlock_bh()
  - ✓Process context and bottom halve share the same data
- spin_lock_nest_lock() & spin_lock_nested()
  - ✓lockdep: Annotate places where we take multiple locks of the same class and avoid deadlock – Commit b7d39aff9145 (“lockdep: spin_lock_nest_lock()”)

# Deadlock between process context and bottom halve (the same core)



# Deadlock between process context and bottom halve (the same core)



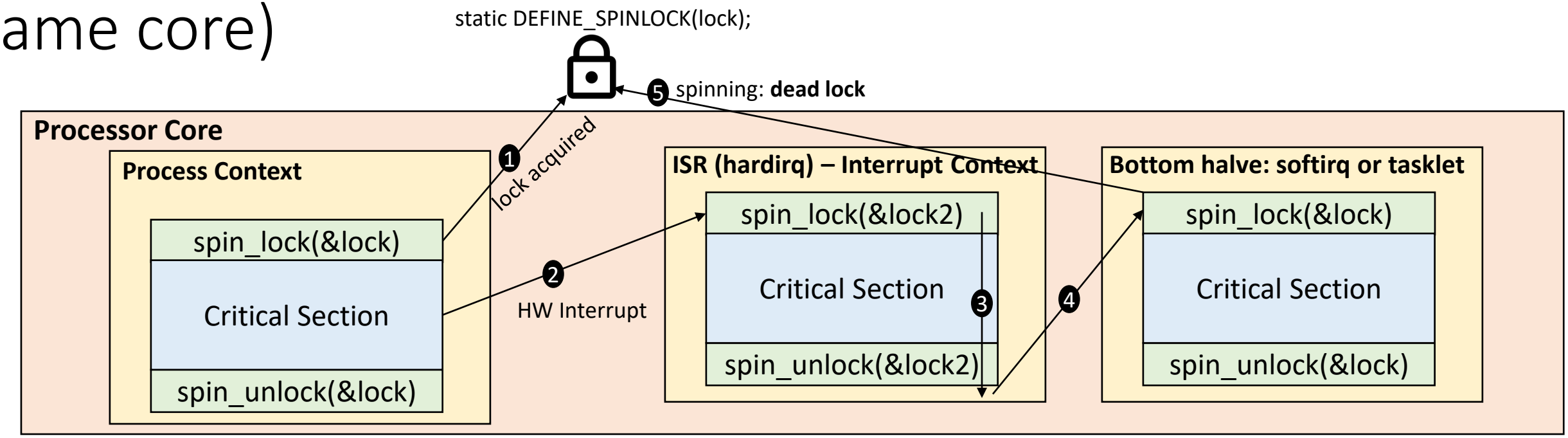

**Bottom halve is invoked when returning from ISR**

```
static inline void __irq_exit_rcu(void)
{
#ifndef __ARCH_IRQ_EXIT_IRQS_DISABLED
        local_irq_disable();
#else
        lockdep_assert_irqs_disabled();
#endif
        account_hardirq_exit(current);
        preempt_count_sub(HARDIRQ_OFFSET);
        if (!in_interrupt() && local_softirq_pending())
                invoke_softirq();

        tick_irq_exit();
}
kernel/softirq.c
```

```
adrian@adrian-ubuntu:linux-5.11$ grep -r __ARCH_IRQ_EXIT_IRQS_DISABLED
kernel/softirq.c:#ifndef __ARCH_IRQ_EXIT_IRQS_DISABLED
arch/arm64/include/asm/hardirq.h:#define __ARCH_IRQ_EXIT_IRQS_DISABLED  1
arch/um/include/asm/hardirq.h:#define __ARCH_IRQ_EXIT_IRQS_DISABLED 1
arch/arm/include/asm/hardirq.h:#define __ARCH_IRQ_EXIT_IRQS_DISABLED    1
arch/powerpc/include/asm/hardirq.h:#define __ARCH_IRQ_EXIT_IRQS_DISABLED
arch/s390/include/asm/hardirq.h:#define __ARCH_IRQ_EXIT_IRQS_DISABLED
```

```
/*
 * The following macros are deprecated and should not be used in new code:
 * in_irq()       - Obsolete version of in_hardirq()
 * in_softirq()   - We have BH disabled, or are processing softirqs
 * in_interrupt() - We're in NMI,IRQ,SoftIRQ context or have BH disabled
 */
#define in_irq()                (hardirq_count())
#define in_softirq()            (softirq_count())
#define in_interrupt()          (irq_count())

include/linux/preempt.h                                    96,1
```

# Deadlock between process context and bottom halve (the same core)




**Softirq is invoked when returning from ISR**

```
static inline void __irq_exit_rcu(void)
{
#ifndef __ARCH_IRQ_EXIT_IRQS_DISABLED
        local_irq_disable();
#else
        lockdep_assert_irqs_disabled();
#endif
        account_hardirq_exit(current);
        preempt_count_sub(HARDIRQ_OFFSET);
        if (!in_interrupt() && local_softirq_pending())
                invoke_softirq();

        tick_irq_exit();
}
```

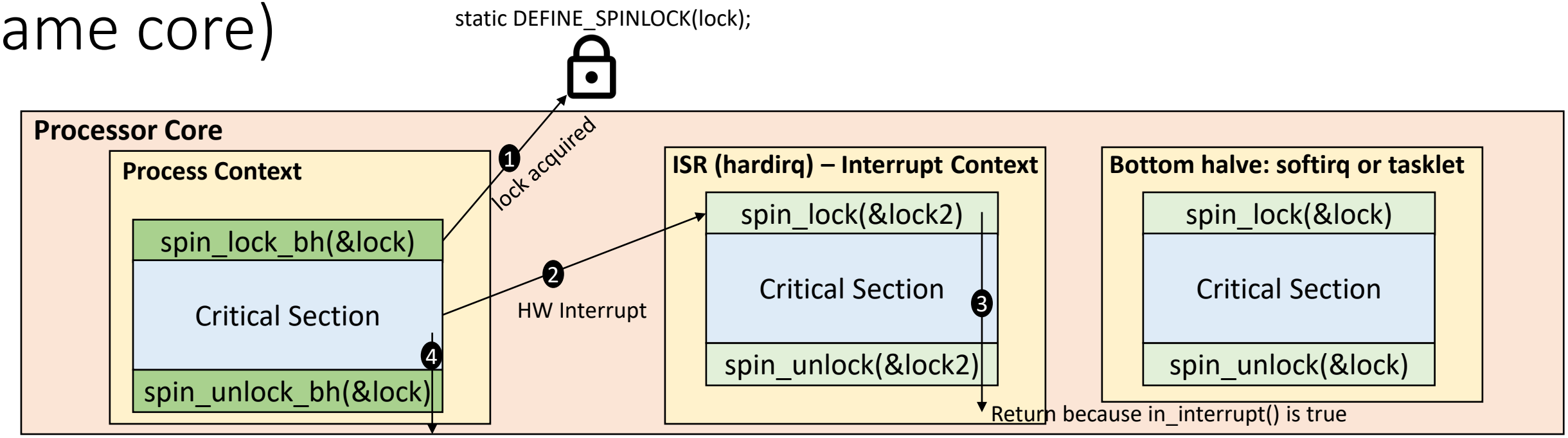
kernel/softirq.c

**Softirq is invoked when running ksoftirqd**

```
static void run_ksoftirqd(unsigned int cpu)
{
        local_irq_disable();
        if (local_softirq_pending()) {
+---  4 lines: We can safely run softirq on
                __do_softirq();
                local_irq_enable();
                cond_resched();
                return;
        }
        local_irq_enable();
}
```

kernel/softirq.c

# Deadlock between process context and bottom halve (the same core)

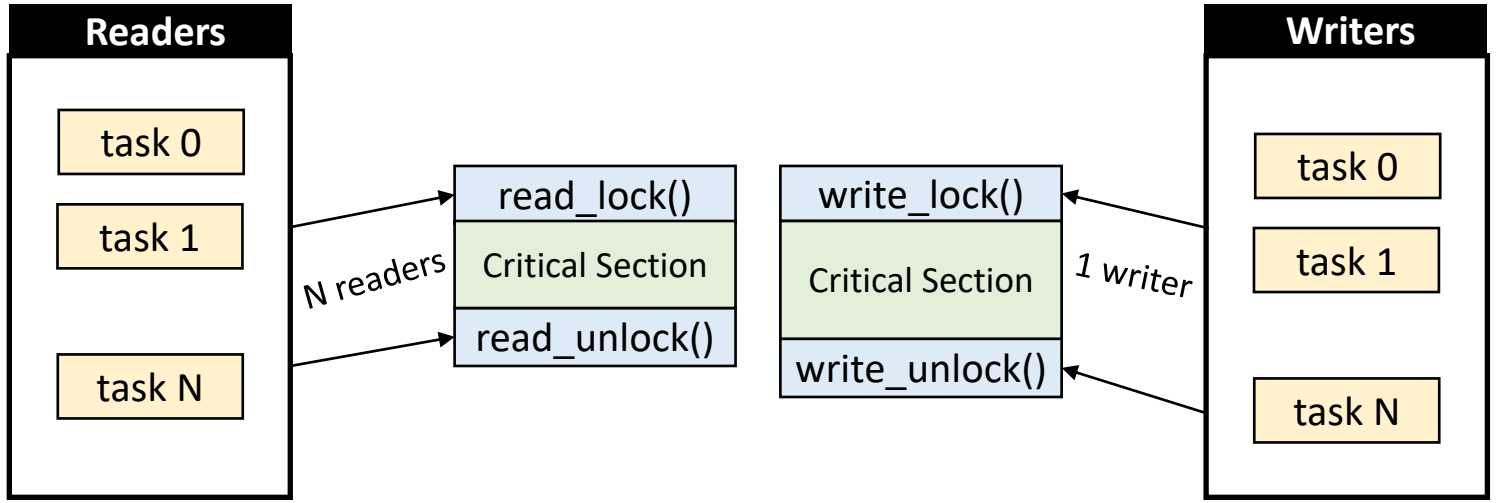



**Softirq is invoked when returning from ISR**

```
static inline void __irq_exit_rcu(void)
{
#ifndef __ARCH_IRQ_EXIT_IRQS_DISABLED
        local_irq_disable();
#else
        lockdep_assert_irqs_disabled();
#endif
        account_hardirq_exit(current);
        preempt_count_sub(HARDIRQ_OFFSET);
        if (!in_interrupt() && local_softirq_pending())
                invoke_softirq();

        tick_irq_exit();
}
```

kernel/softirq.c

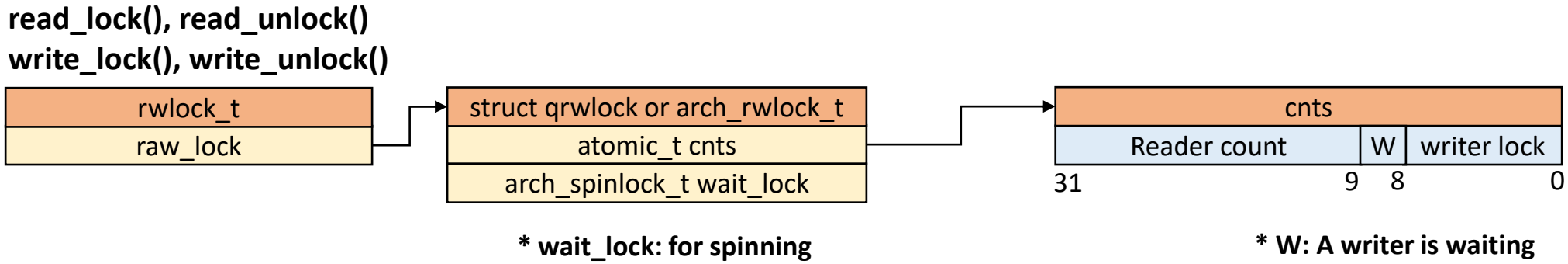
# rwlock (reader-writer spinlock)



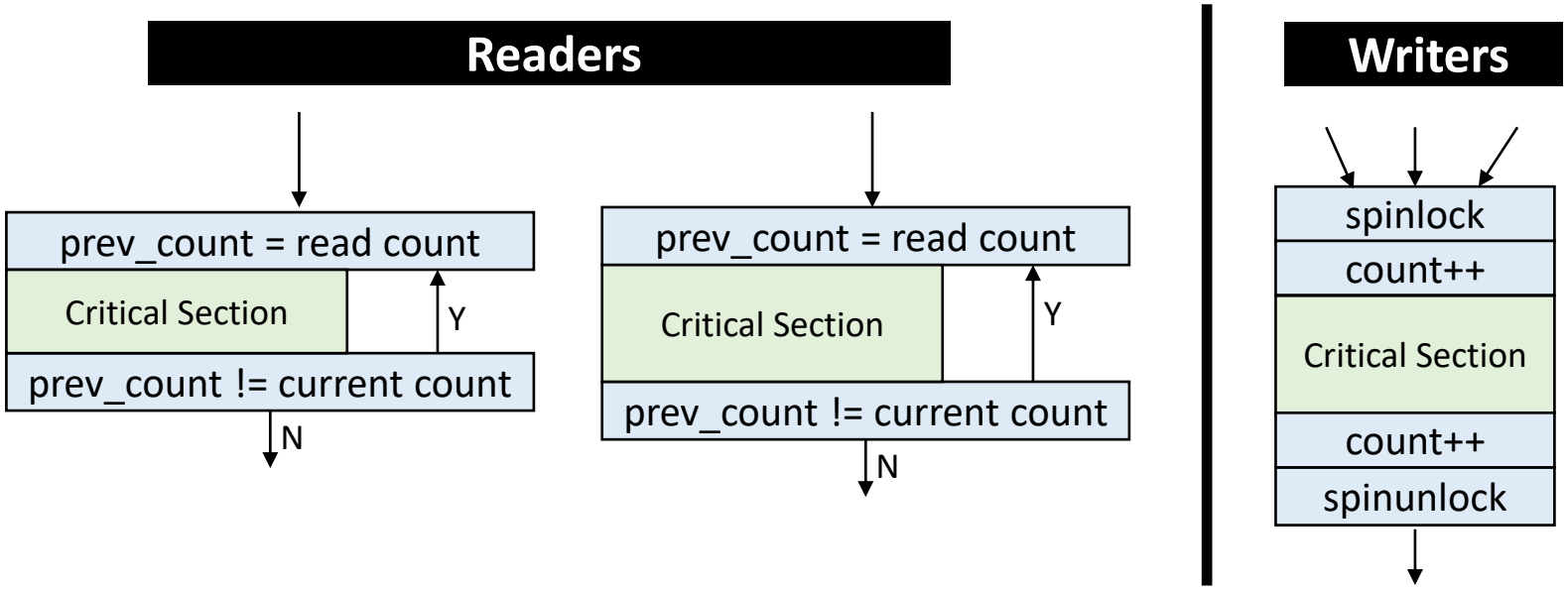

- Concept
  - ✓ [Without readers] Only one writer can enter CS
  - ✓ [Without writer] N-readers can enter CS simultaneously
  - ✓ Readers and writer can enter CS simultaneously
  - ✓ Reader(s) in CS → The writer needs to wait (spinning)
  - ✓ Writer in CS → Readers need to wait (spinning)
- Mutual exclusion between reader and writer
- Writer might be starved!
  - Reader has higher priority than writer
- Useful scenario: search for linked list without changing the list
  - ✓ Example: tasklist_lock → __cacheline_aligned DEFINE_RWLOCK(tasklist_lock);
- Linux kernel developers are trying to remove rwlock in most cases.

[Reference] Lesson 2: reader-writer spinlocks

# rwlock (reader-writer spinlock): Data structure



# seqlock (Sequential lock)




- Lockless readers (only retry loop)
- No writer starvation
  - ✓ Writer has higher priority than reader
  - ✓ No mutual exclusion between reader and writer
    - ➢ Mutual exclusion between writers because of writers' spinlock
- Scenario: lots of readers and few writers
  - ✓ Example: jiffies → Check get_jiffies_64(), tick_setup_periodic() and so on.

# spinlock derivative

| | spinlock | rwlock | seqlock |
|---|---|---|---|
| **Reader(s)** | 1 | N | N |
| **Writer(s)** | 1 | 1 | 1 |

# spinlock derivative and RCU

| | spinlock | rwlock | seqlock | RCU |
|---|---|---|---|---|
| **Reader(s)** | 1 | N | N | N |
| **Writer(s)** | 1 | 1 | 1 | 1 |

# Reference


- Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." Proceedings of the Linux Symposium. 2012. http://pdos.csail.mit.edu/papers/linux:lock.pdf
- T. E. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors", IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pp. 6-16, 1990.
- J. M. Mellor-Crummey and M. L. Scott. “Algorithms for scalable synchronization on shared-memory multiprocessors”, ACM Transactions on Computer Systems, 9(1):21–65, 1991.
- MCS locks and qspinlocks, LWN
- Is Parallel Programming Hard, And, If So, What Can You Do About It?