# Linux Synchronization Mechanism: Semaphore & Mutex

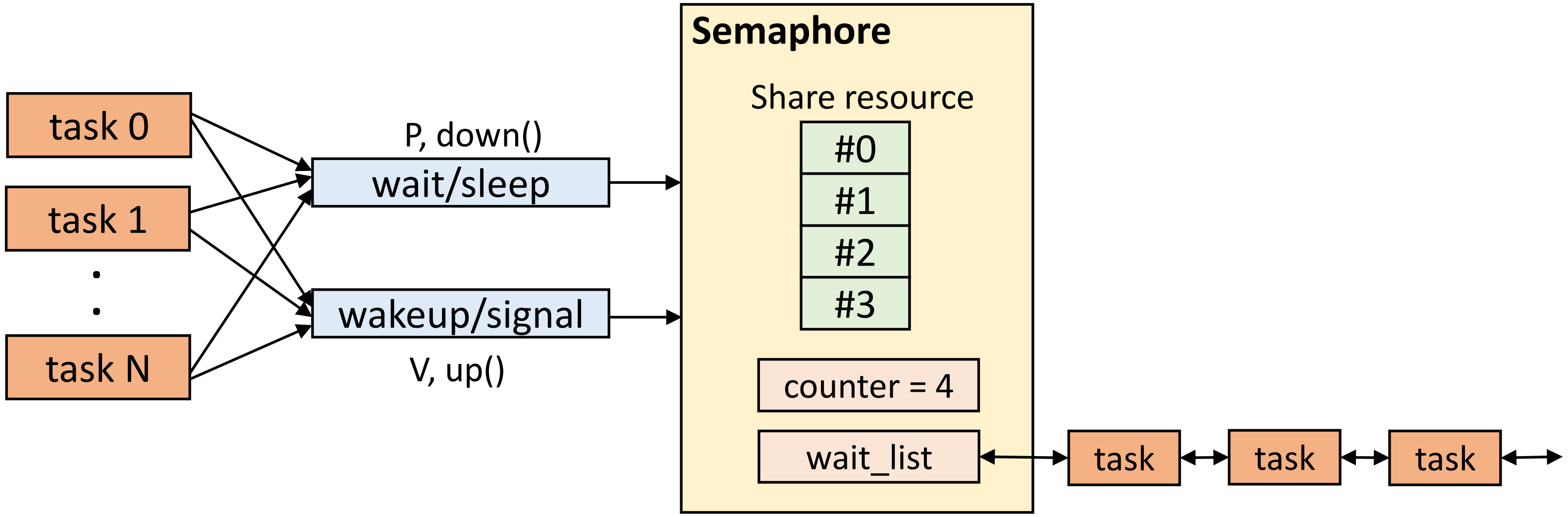
Adrian Huang | Feb, 2023

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

# Agenda

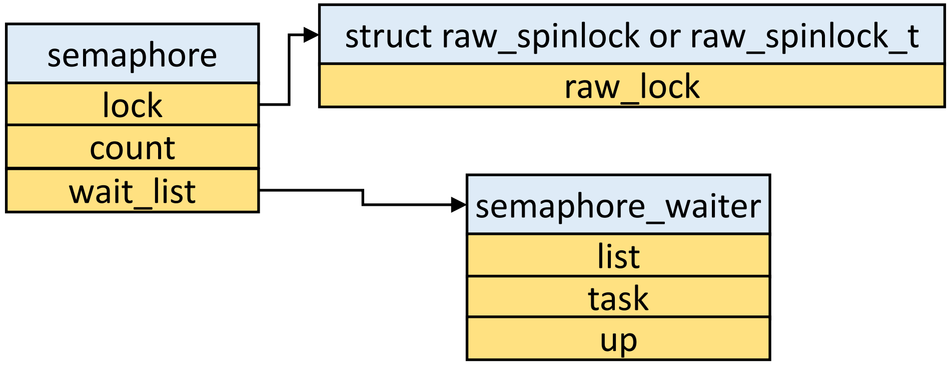
- Semaphore
  - ✓producer-consumer problem
  - ✓Implementation in Linux kernel
- Mutex (introduced in v2.6.16)
  - ✓Enforce serialization on shared memory systems
  - ✓Implementation in Linux kernel
  - ✓Mutex lock
    - ➢ Fast path, midpath, slow path
  - ✓Mutex unlock
    - ➢ Fast path and slow path
    - ➢ Mutex ownership (with a lab)
      - ■ Re-visit this concept: Only the lock owner has the permission to unlock the mutex
  - ✓Q & A

# Semaphore: producer-consumer problem



- Sleeping lock
- Used in process context *ONLY*
- Cannot hold a spin lock while acquiring a semaphore
- Mainly use in producer-consumer scenario
- The lock holder does not require to unlock the lock. (non-ownership concept)
  - ✓ Something like notification

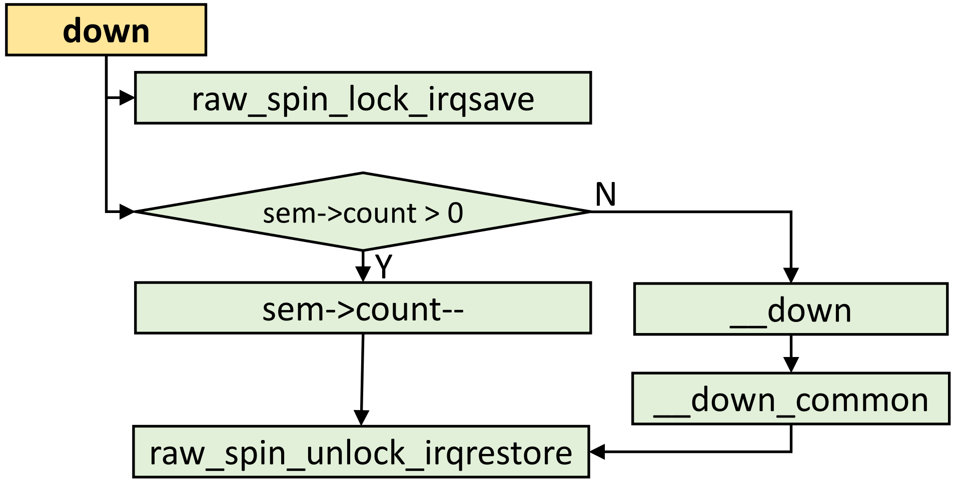
# Semaphore Implementation in Linux Kernel



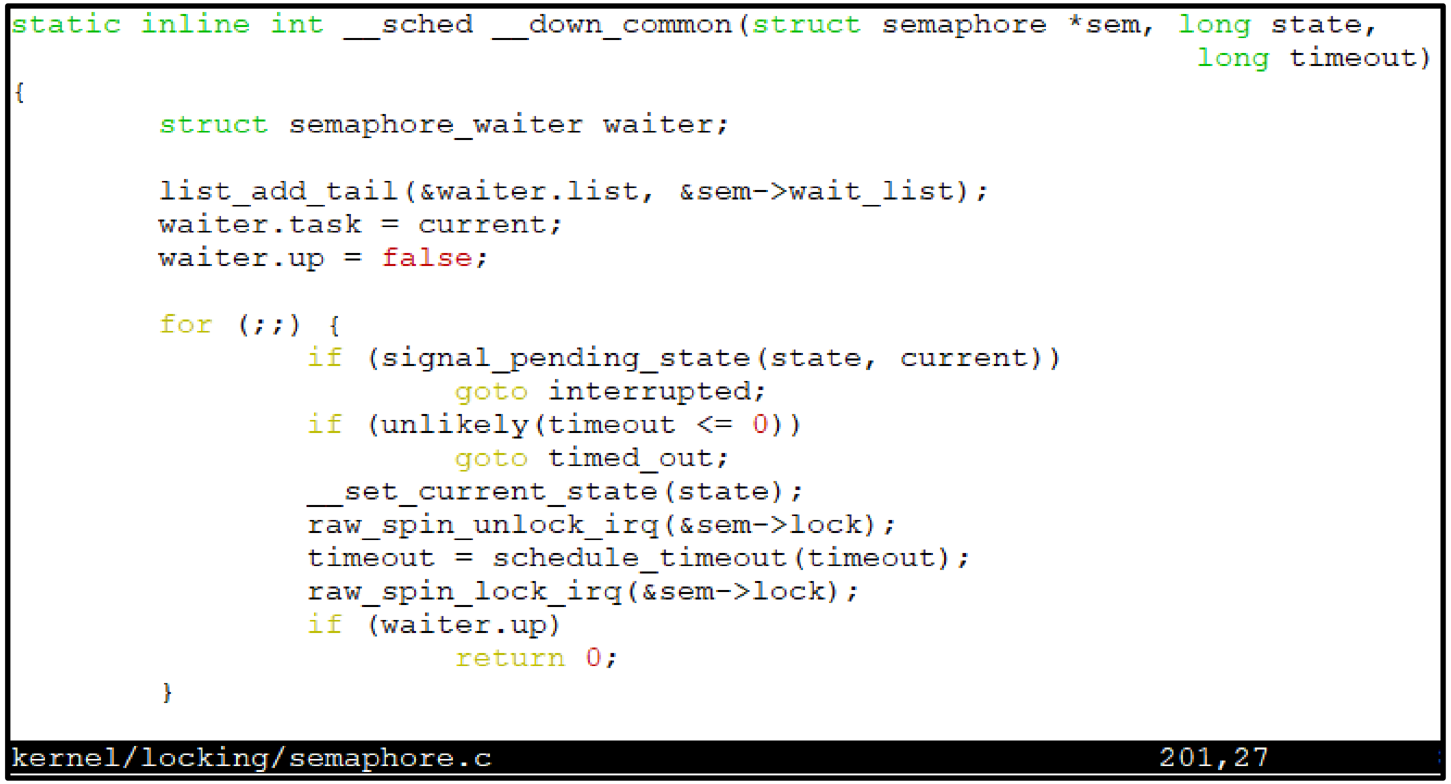
semaphore
lock
count
wait_list

struct raw_spinlock or raw_spinlock_t
raw_lock

semaphore_waiter
list
task
up

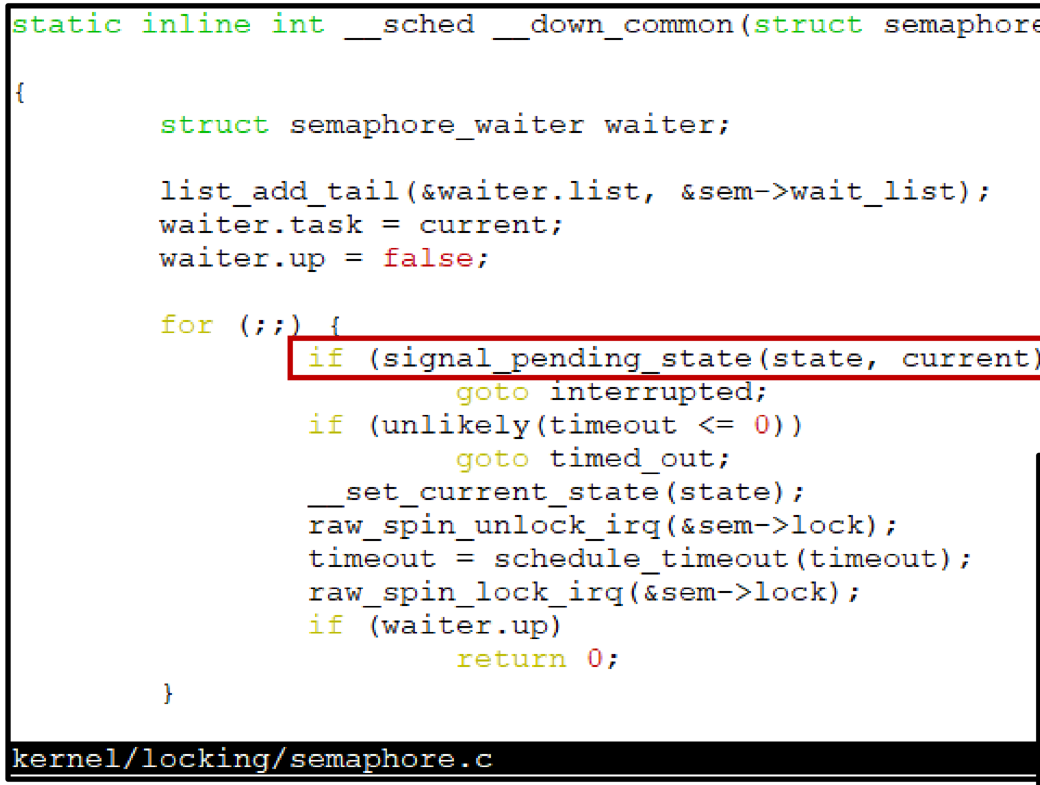


down
raw_spin_lock_irqsave
sem->count > 0
N
Y
sem->count--
__down
__down_common
raw_spin_unlock_irqrestore

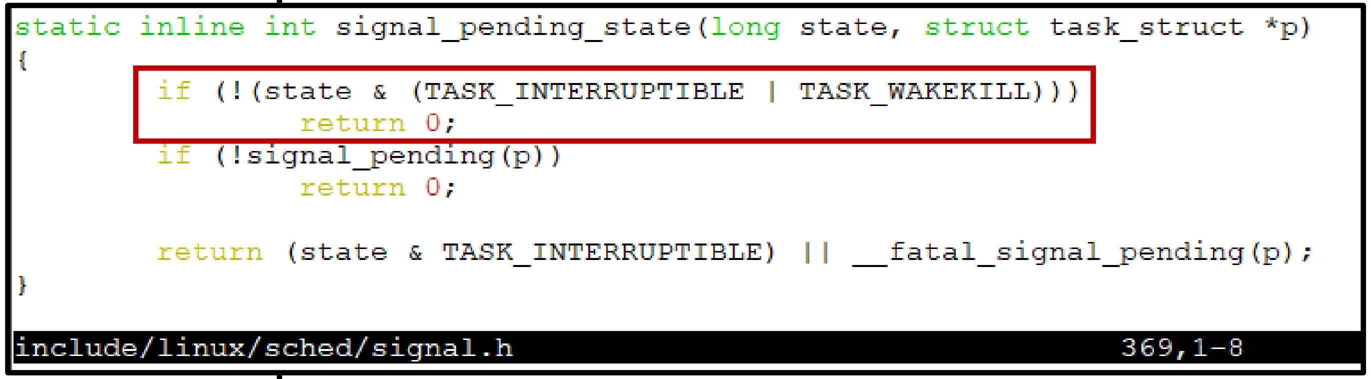


```
static inline int __sched __down_common(struct semaphore *sem, long state,
                                                                long timeout)
{
        struct semaphore_waiter waiter;

        list_add_tail(&waiter.list, &sem->wait_list);
        waiter.task = current;
        waiter.up = false;

        for (;;) {
                if (signal_pending_state(state, current))
                        goto interrupted;
                if (unlikely(timeout <= 0))
                        goto timed_out;
                __set_current_state(state);
                raw_spin_unlock_irq(&sem->lock);
                timeout = schedule_timeout(timeout);
                raw_spin_lock_irq(&sem->lock);
                if (waiter.up)
                        return 0;
        }
kernel/locking/semaphore.c                                          201,27
```

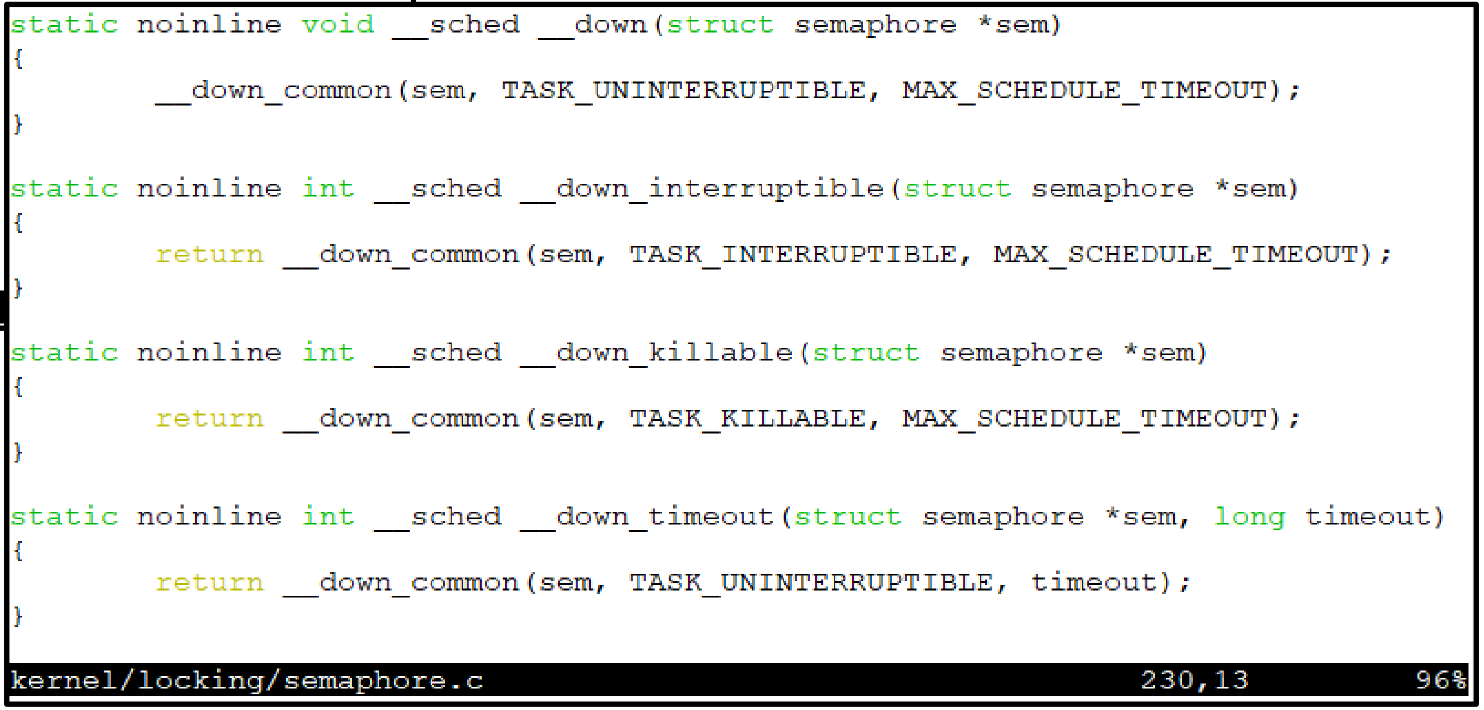
# Semaphore Implementation in Linux Kernel



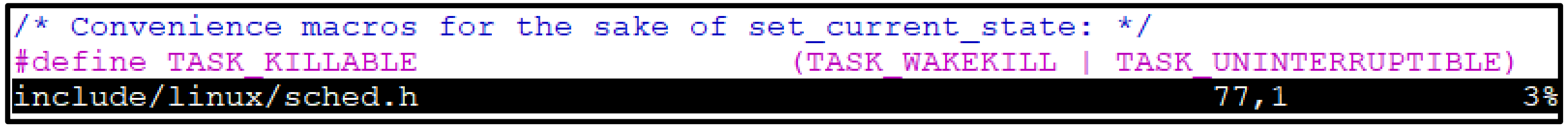
```
static inline int __sched __down_common(struct semaphore *sem, long state,
                                long timeout)
{
        struct semaphore_waiter waiter;

        list_add_tail(&waiter.list, &sem->wait_list);
        waiter.task = current;
        waiter.up = false;

        for (;;) {
                if (signal_pending_state(state, current))
                        goto interrupted;
                if (unlikely(timeout <= 0))
                        goto timed_out;
                __set_current_state(state);
                raw_spin_unlock_irq(&sem->lock);
                timeout = schedule_timeout(timeout);
                raw_spin_lock_irq(&sem->lock);
                if (waiter.up)
                        return 0;
        }
kernel/locking/semaphore.c
```



```
static inline int signal_pending_state(long state, struct task_struct *p)
{
        if (!(state & (TASK_INTERRUPTIBLE | TASK_WAKEKILL)))
                return 0;
        if (!signal_pending(p))
                return 0;

        return (state & TASK_INTERRUPTIBLE) || __fatal_signal_pending(p);
}
include/linux/sched/signal.h                                        369,1-8
```

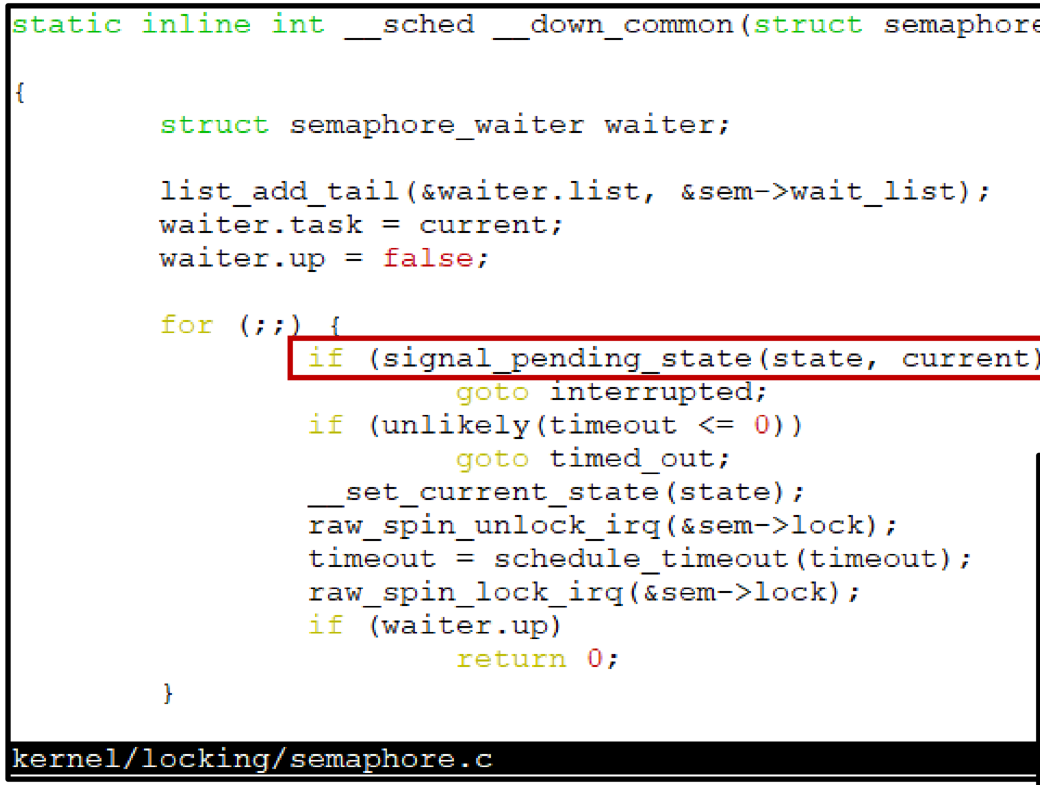


```
static noinline void __sched __down(struct semaphore *sem)
{
        __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_interruptible(struct semaphore *sem)
{
        return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_killable(struct semaphore *sem)
{
        return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_timeout(struct semaphore *sem, long timeout)
{
        return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
kernel/locking/semaphore.c                                    230,13         96%
```

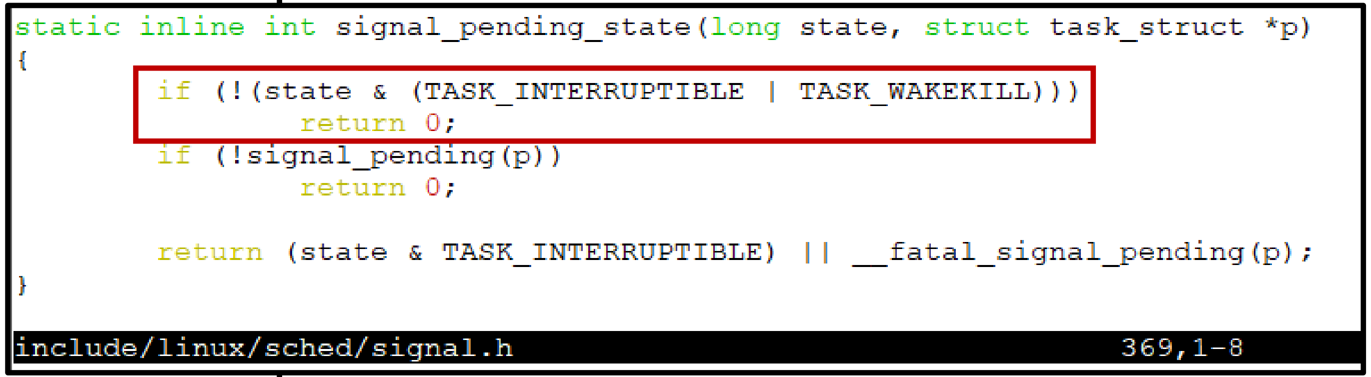


```
/* Convenience macros for the sake of set_current_state: */
#define TASK_KILLABLE                   (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
include/linux/sched.h                                           77,1           3%
```

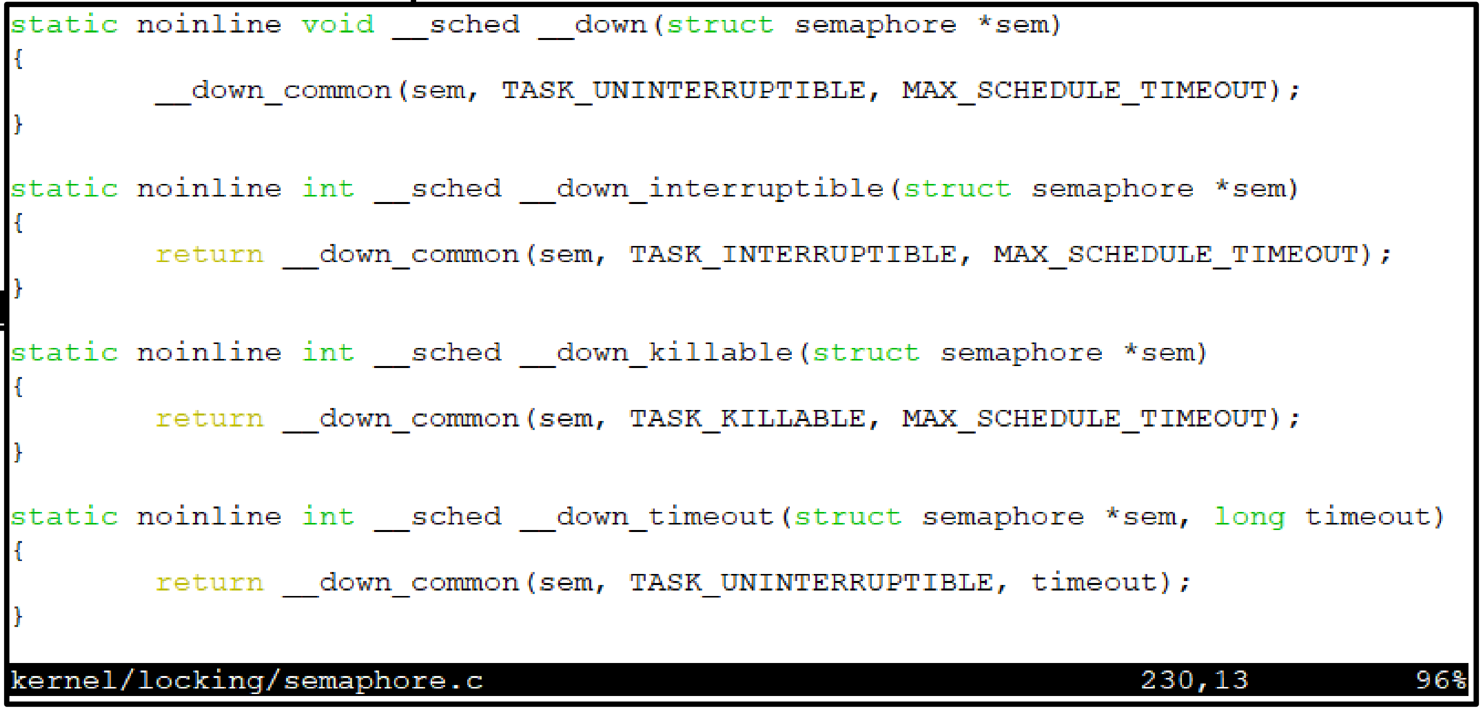
# Semaphore Implementation in Linux Kernel



```
static inline int __sched __down_common(struct semaphore *sem, long state,
                                         long timeout)
{
        struct semaphore_waiter waiter;

        list_add_tail(&waiter.list, &sem->wait_list);
        waiter.task = current;
        waiter.up = false;

        for (;;) {
                if (signal_pending_state(state, current))
                        goto interrupted;
                if (unlikely(timeout <= 0))
                        goto timed_out;
                __set_current_state(state);
                raw_spin_unlock_irq(&sem->lock);
                timeout = schedule_timeout(timeout);
                raw_spin_lock_irq(&sem->lock);
                if (waiter.up)
                        return 0;
        }
kernel/locking/semaphore.c
```



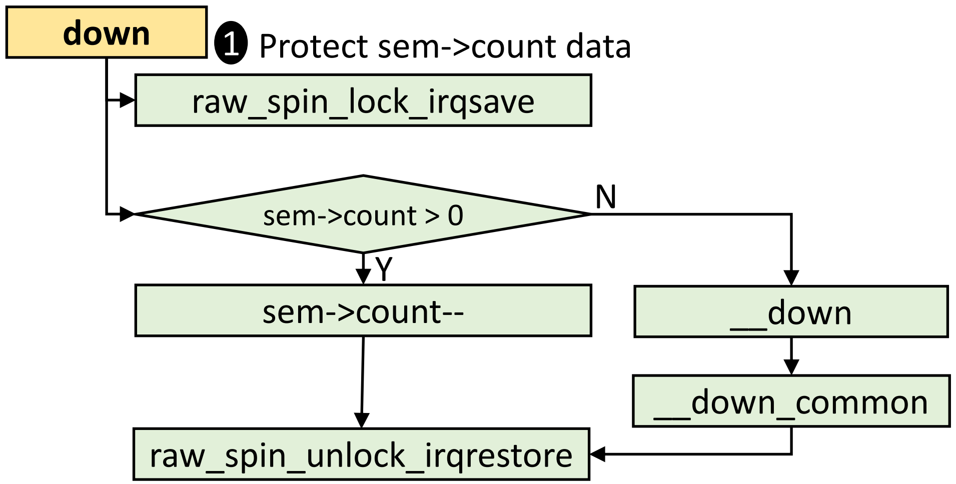
```
static inline int signal_pending_state(long state, struct task_struct *p)
{
        if (!(state & (TASK_INTERRUPTIBLE | TASK_WAKEKILL)))
                return 0;
        if (!signal_pending(p))
                return 0;

        return (state & TASK_INTERRUPTIBLE) || __fatal_signal_pending(p);
}
include/linux/sched/signal.h                                369,1-8
```



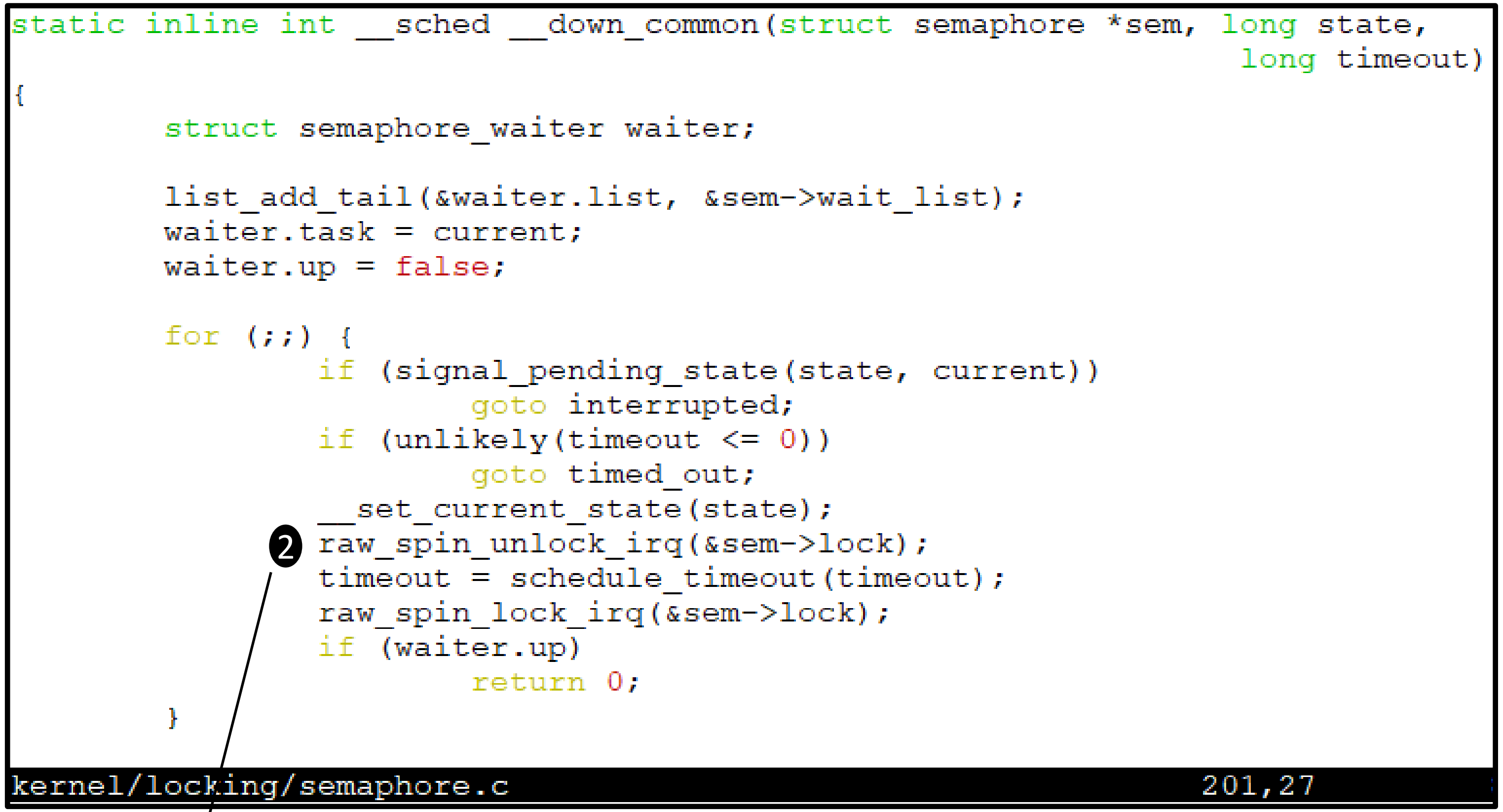
```
static noinline void __sched __down(struct semaphore *sem)
{
        __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_interruptible(struct semaphore *sem)
{
        return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_killable(struct semaphore *sem)
{
        return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_timeout(struct semaphore *sem, long timeout)
{
        return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
kernel/locking/semaphore.c                            230,13          96%
```

**[Only for interruptible and wakekill task] Check if the sleeping task gets a signal**

# Semaphore Implementation in Linux Kernel



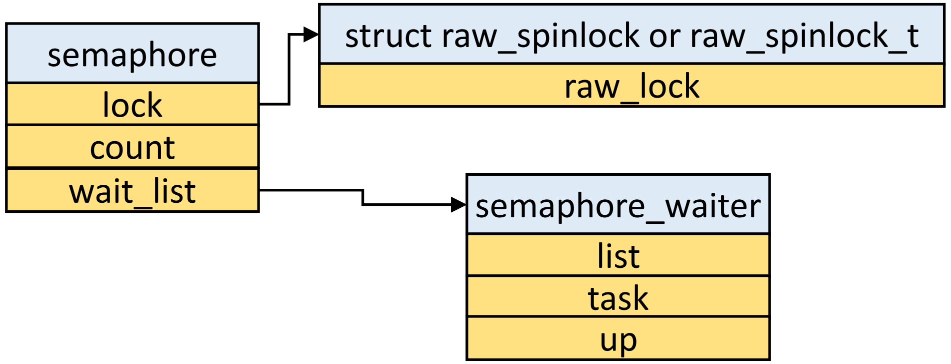
1 Protect sem->count data

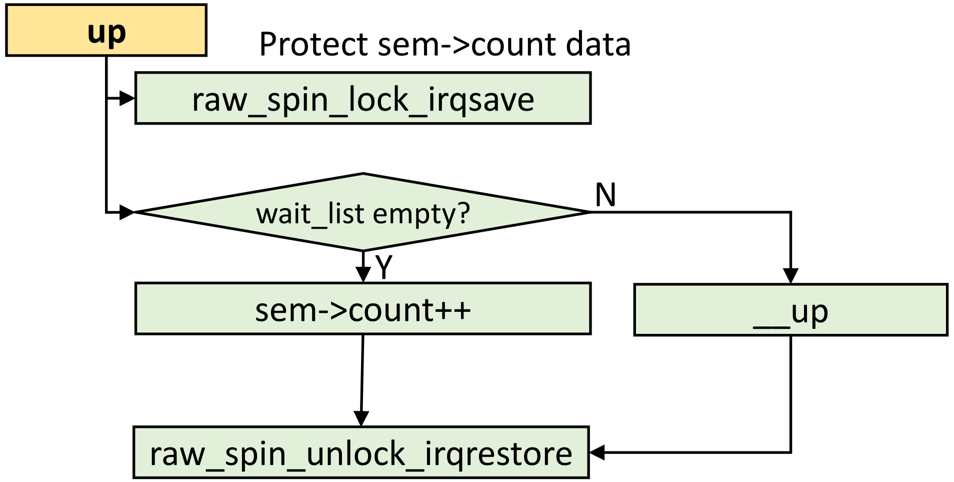


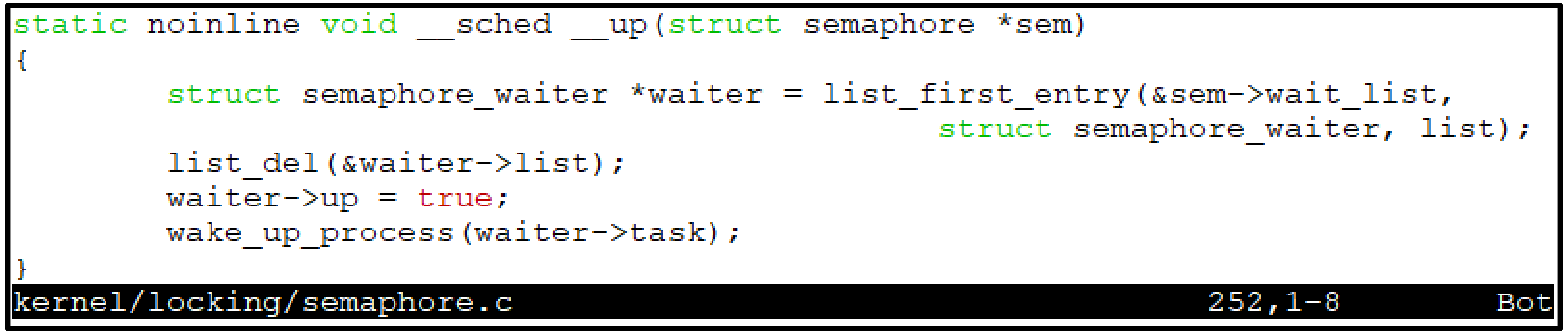
```
static inline int __sched __down_common(struct semaphore *sem, long state,
                                                                long timeout)
{
        struct semaphore_waiter waiter;

        list_add_tail(&waiter.list, &sem->wait_list);
        waiter.task = current;
        waiter.up = false;

        for (;;) {
                if (signal_pending_state(state, current))
                        goto interrupted;
                if (unlikely(timeout <= 0))
                        goto timed_out;
                __set_current_state(state);
                raw_spin_unlock_irq(&sem->lock);
                timeout = schedule_timeout(timeout);
                raw_spin_lock_irq(&sem->lock);
                if (waiter.up)
                        return 0;
        }
kernel/locking/semaphore.c                                          201,27
```

2 Reschedule: Need to unlock spinlock

# Semaphore Implementation in Linux Kernel





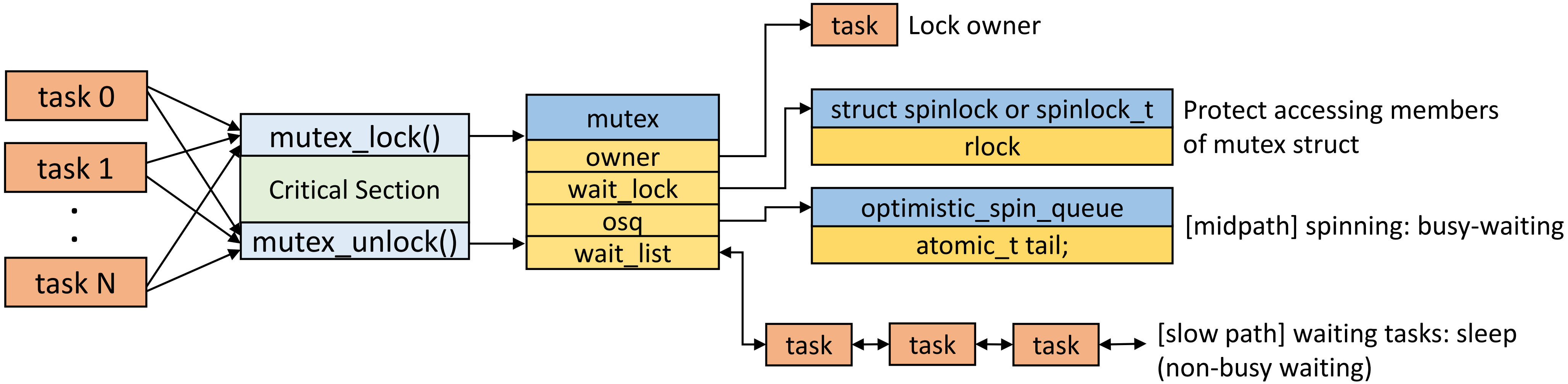


```
static noinline void __sched __up(struct semaphore *sem)
{
        struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
                                                struct semaphore_waiter, list);
        list_del(&waiter->list);
        waiter->up = true;
        wake_up_process(waiter->task);
}
kernel/locking/semaphore.c                                   252,1-8        Bot
```

# Agenda

- Semaphore
  - ✓producer-consumer problem
  - ✓Implementation in Linux kernel
- Mutex (introduced in v2.6.16)
  - ✓Enforce serialization on shared memory systems
  - ✓Implementation in Linux kernel
  - ✓Call path
    - ➢ Fast path, midpath, slow path

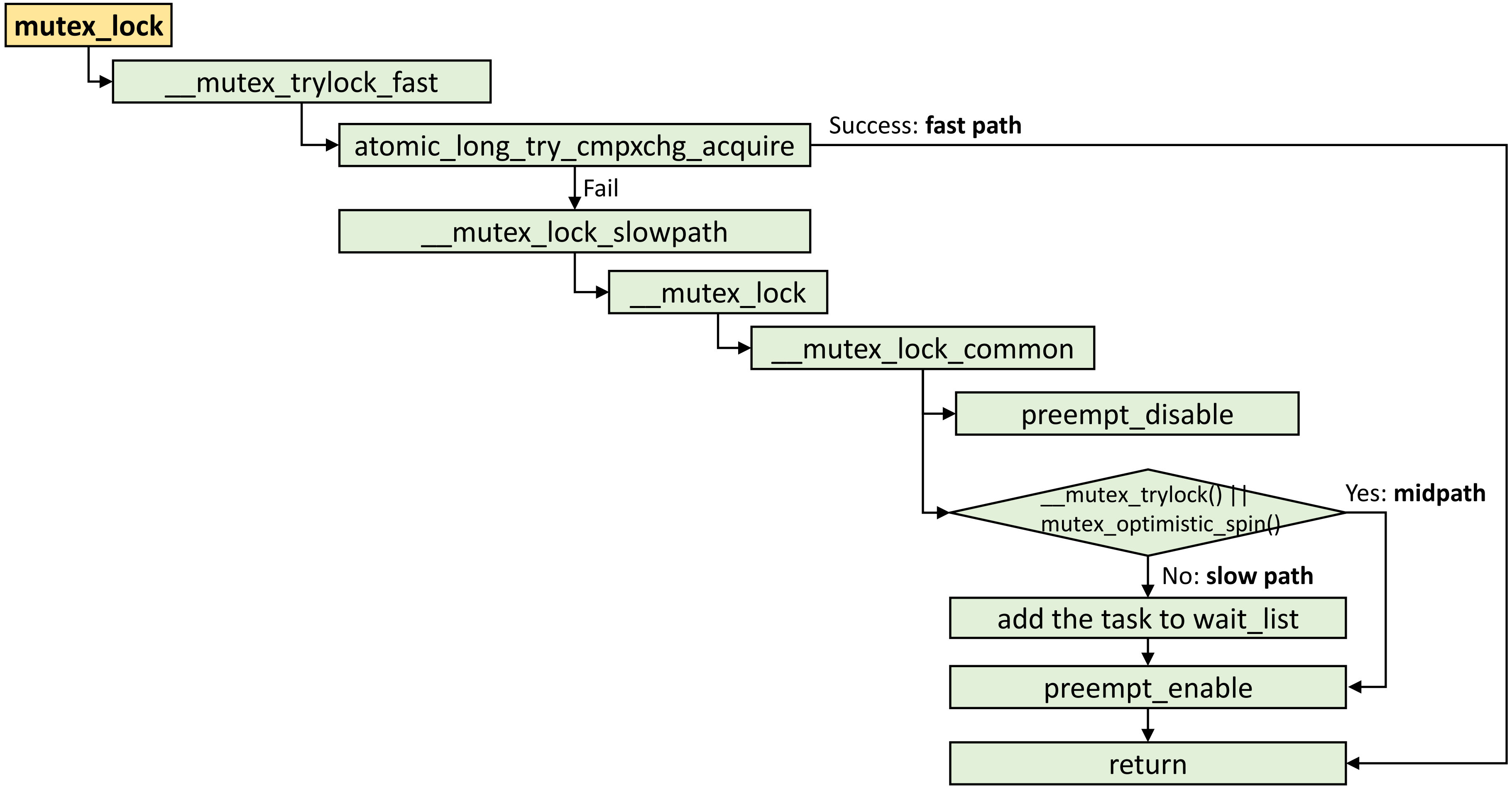
# Mutex: Enforce serialization on shared memory systems



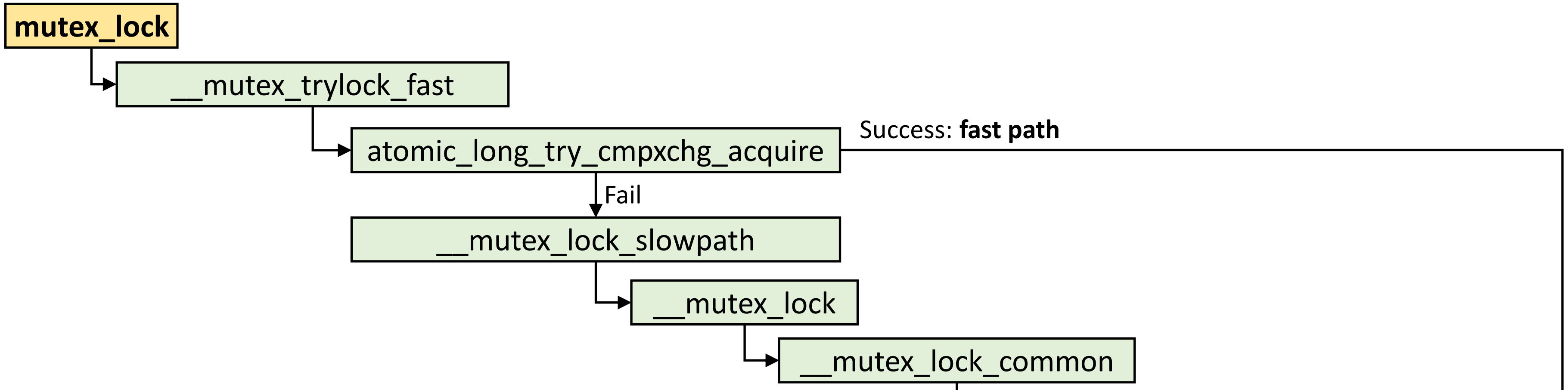
# Mutex Implementation in Linux

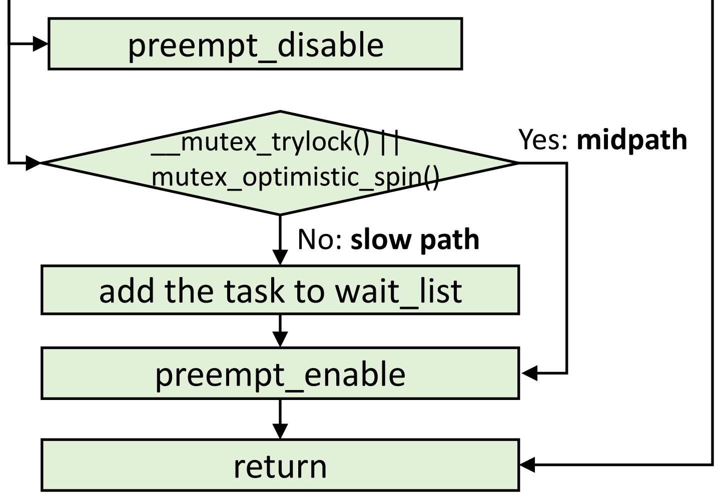
- Mutex implementation paths
  - ✓Fastpath: Uncontended case by using cmpxchg(): CAS (Compare and Swap)
  - ✓Midpath (optimistic spinning) - The priority of the lock owner is the highest one
    - ➢Spin for mutex lock acquisition when the lock owner is running.
    - ➢The lock owner is likely to release the lock soon.
    - ➢Leverage cancelable MCS lock (OSQ - Optimistic Spin Queue: MCS-like lock): v3.15
  - ✓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
- Ownership: Only the lock owner can release the lock
- kernel/locking/{mutex.c, osq_lock.c}
- Reference: Generic Mutex Subsystem

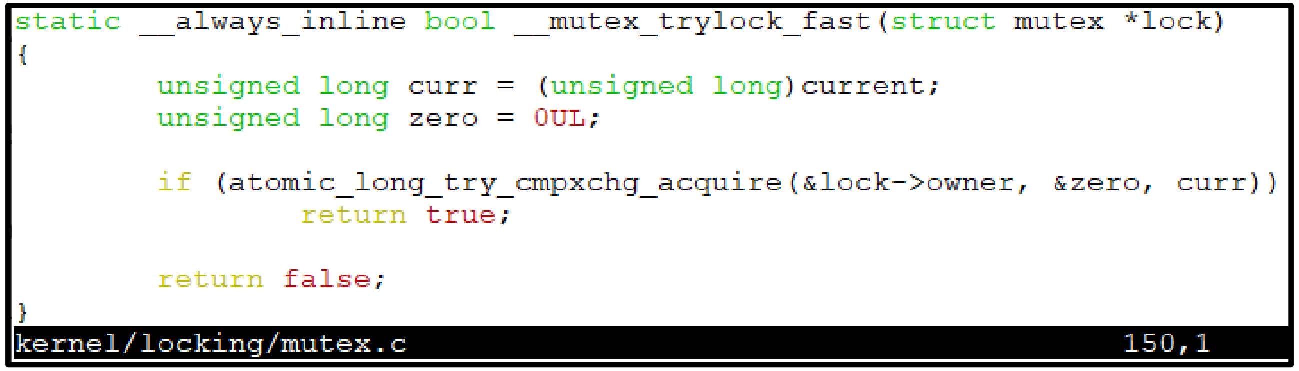
# mutex_lock(): Call path



# mutex_lock(): Fast path

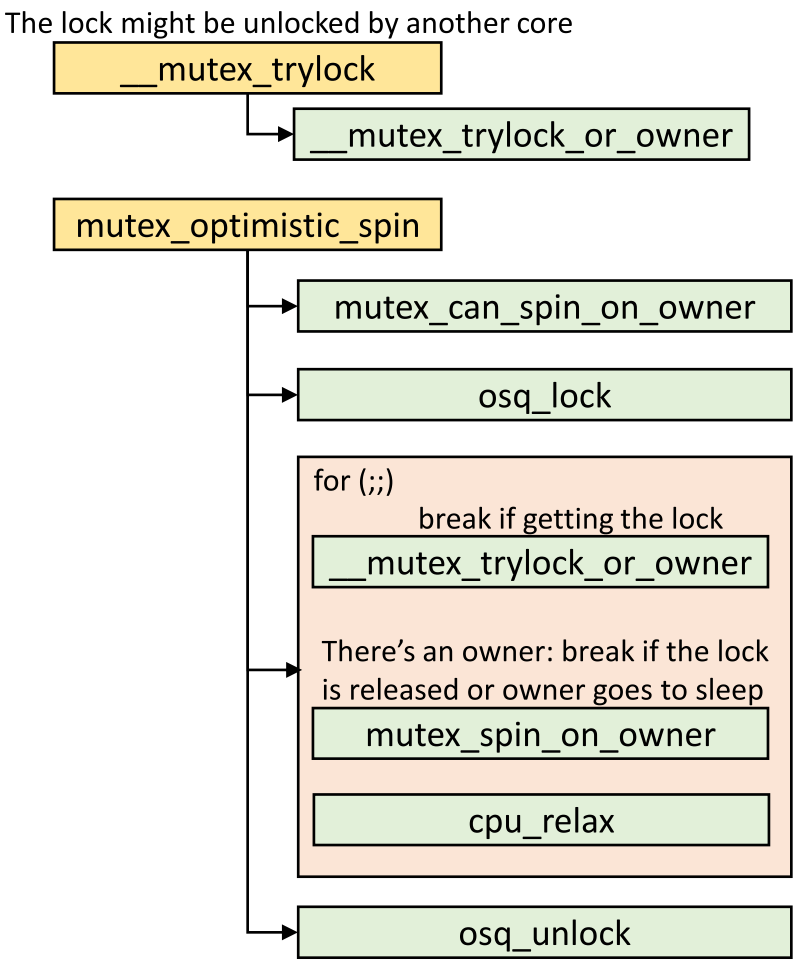






```
static __always_inline bool __mutex_trylock_fast(struct mutex *lock)
{
        unsigned long curr = (unsigned long)current;
        unsigned long zero = 0UL;

        if (atomic_long_try_cmpxchg_acquire(&lock->owner, &zero, curr))
                return true;

        return false;
}
kernel/locking/mutex.c                                       150,1
```

# mutex_lock(): midpath



The lock might be unlocked by another core

## mutex_can_spin_on_owner

**Return true if the following conditions are met**

- The spinning task is not preempted: need_resched()
- The lock owner:
  - ✓ Not preempted : checked by vcpu_is_preempted()
  - ✓ Not sleep: checked by owner->on_cpu
- Spinner is spinning on the current lock owner!

## mutex_spin_on_owner

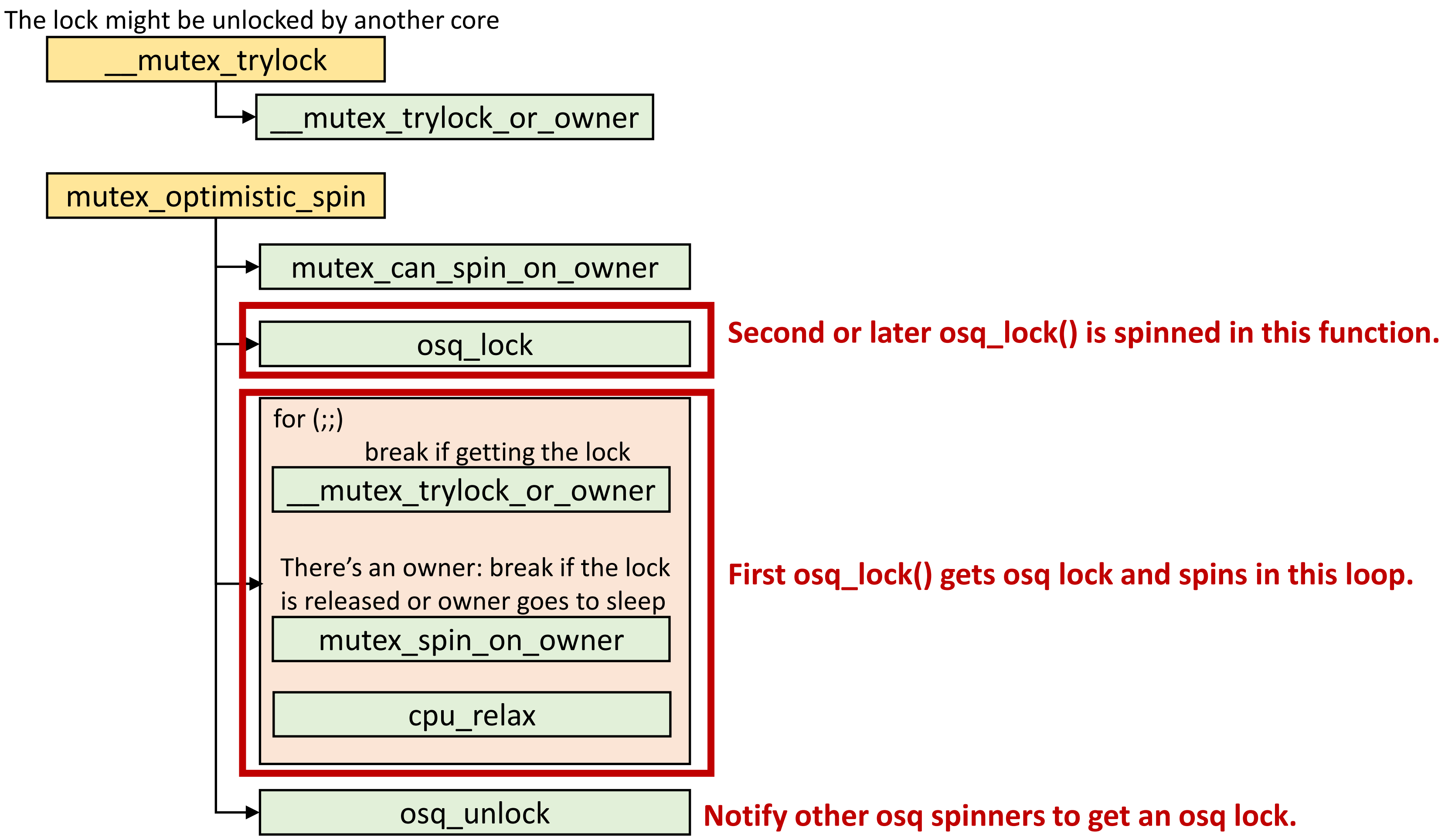
**mutex_spin_on_owner() returns true → keep looping for acquiring the lock**

- Lock release: one of spinning tasks can get the lock

**mutex_spin_on_owner() returns false → break ‘for’ loop**

- The spinning task is preempted
- The lock owner is preempted
- The lock owner sleeps

# mutex_lock(): midpath

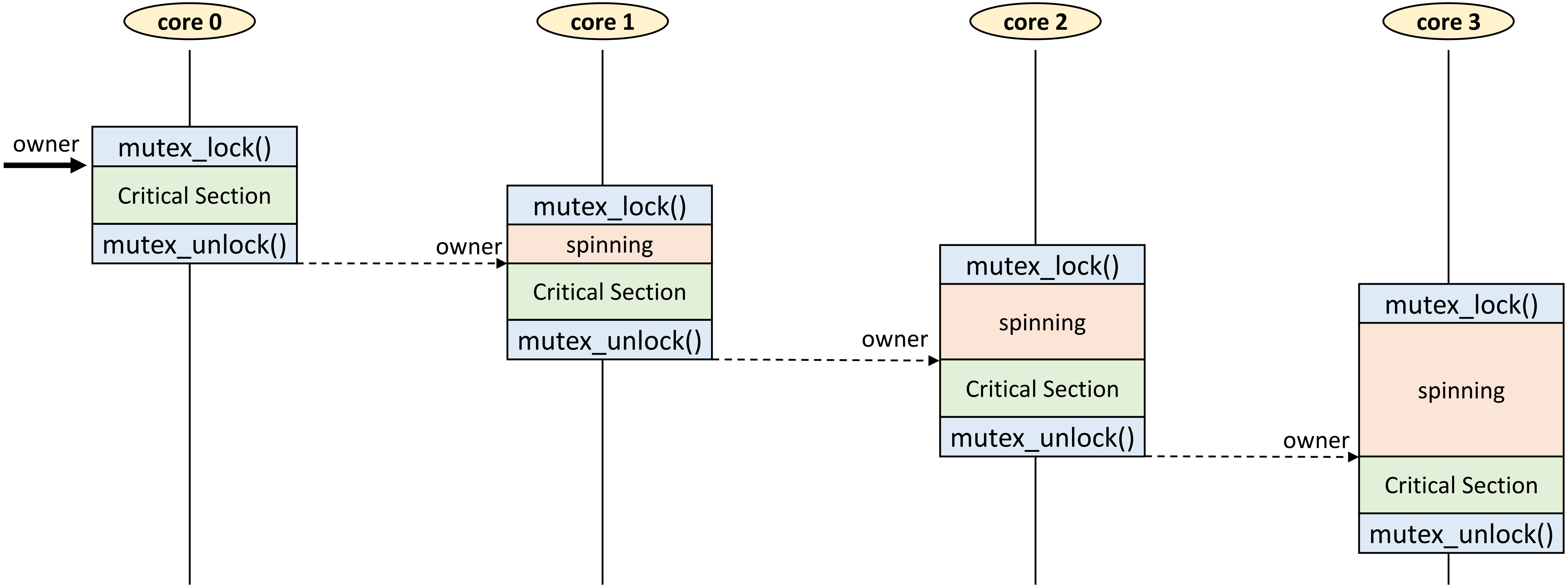


Second or later osq_lock() is spinned in this function.

First osq_lock() gets osq lock and spins in this loop.

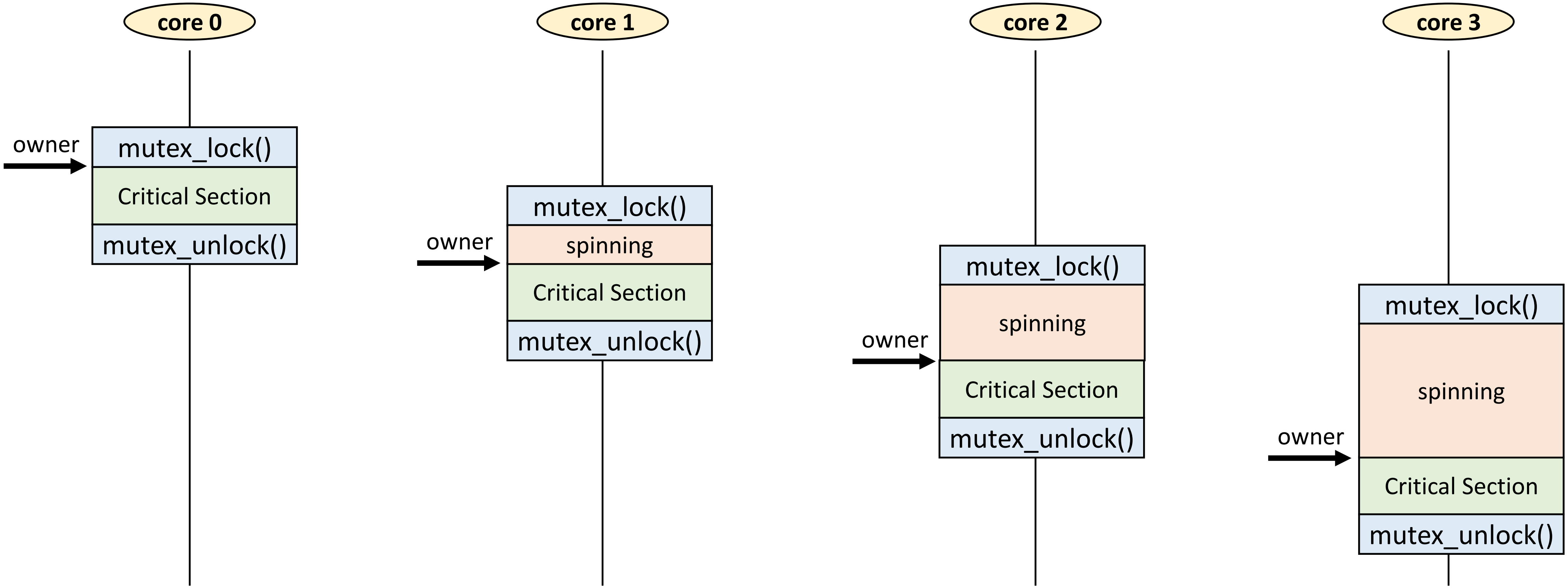
Notify other osq spinners to get an osq lock.

# midpath: [Case #1: ideal] without preemption or sleep (both lock owner and spinner)



**One of spinning tasks can get the lock after the owner releases the lock: Spinning tasks do not need to be moved to wait list**

# midpath – [Case #1: ideal] lock release without preemption or sleep



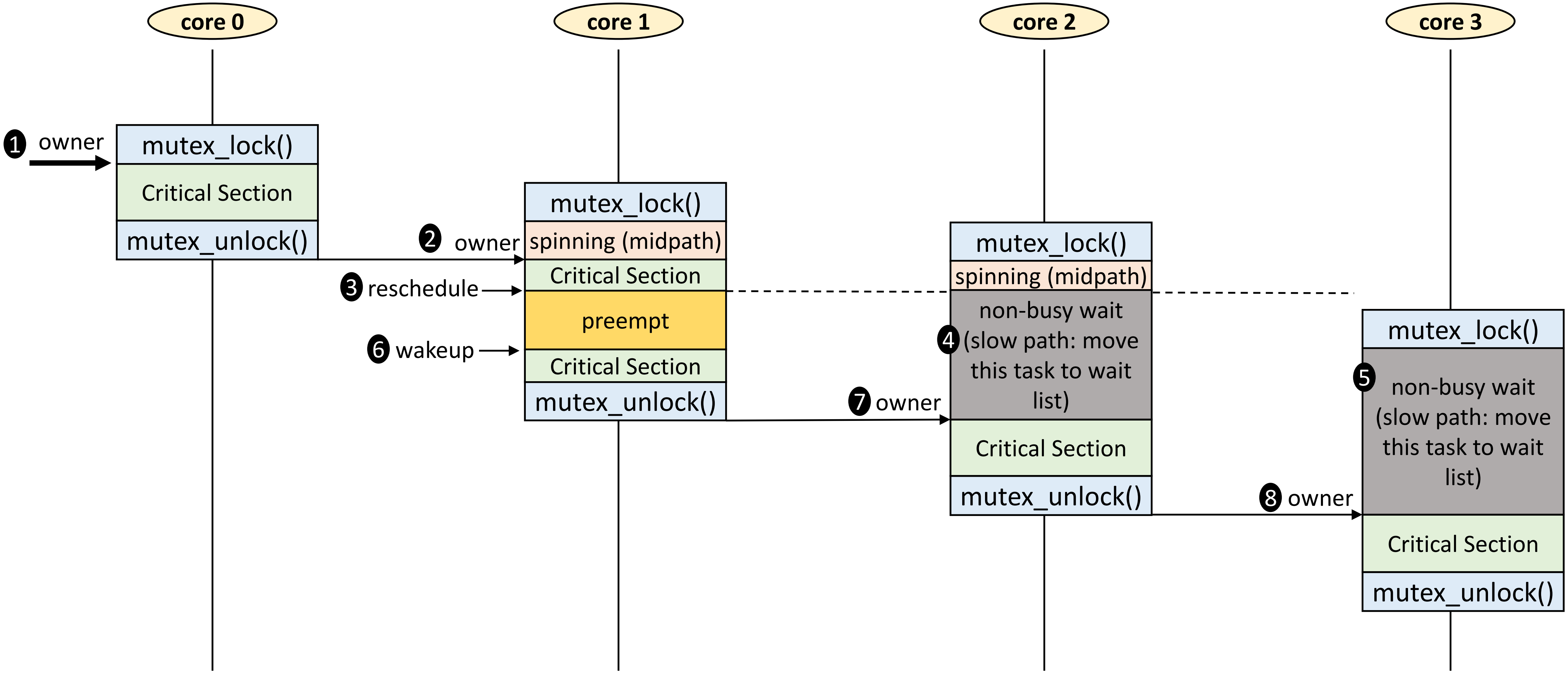
**When to exit the spinning?**

1. The lock owner releases the lock
2. The lock owner goes to sleep or is preempted: spinning tasks go to slow path
   - ✓ Check task->on_cpu
   - ✓ Functions: prepare_task(), finish_task()…
3. The spinning task is preempted: the spinning task goes to slow path
   - ✓ need_resched()

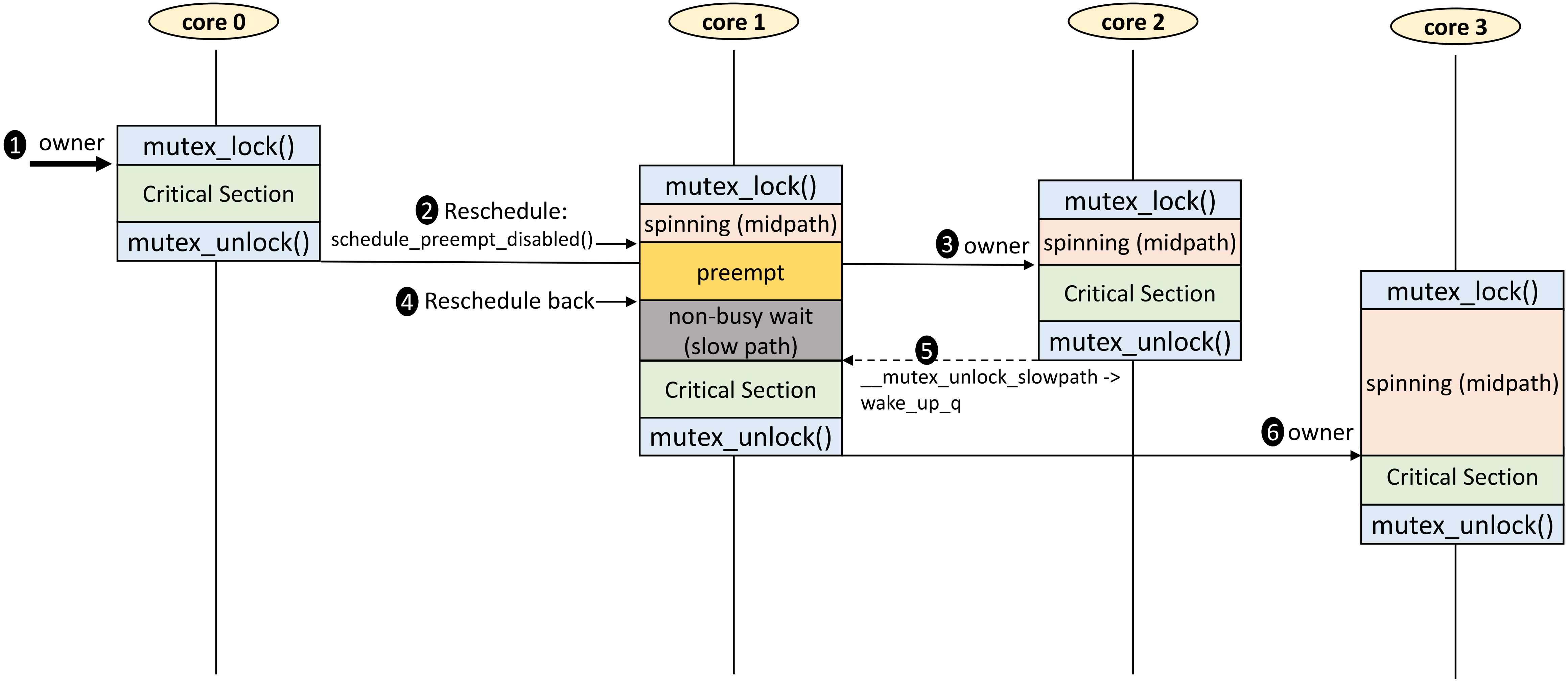
# Three cases for “cannot spin on mutex owner”

- The lock owner is preempted
- The spinning task is preempted
- The lock owner sleeps

# midpath: [Case #2] Mutex lock owner is preempted



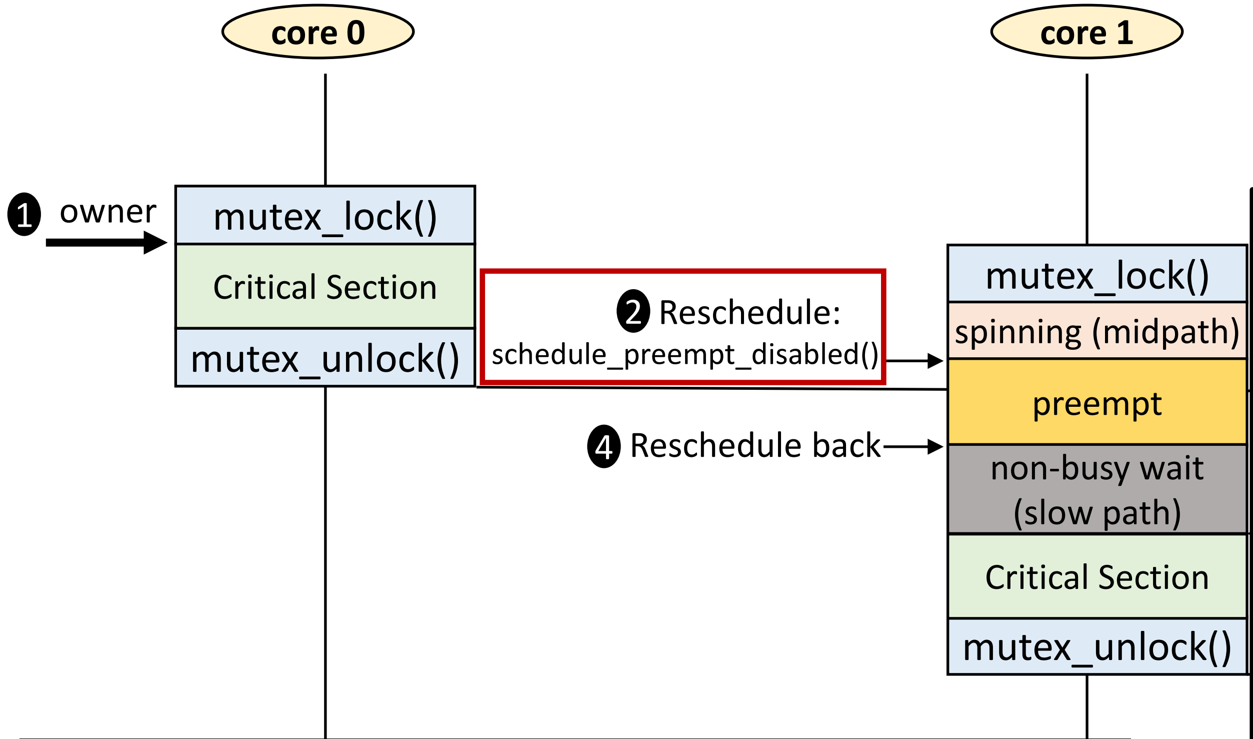
# midpath: [Case #3] Spinner (osq lock owner) is preempted

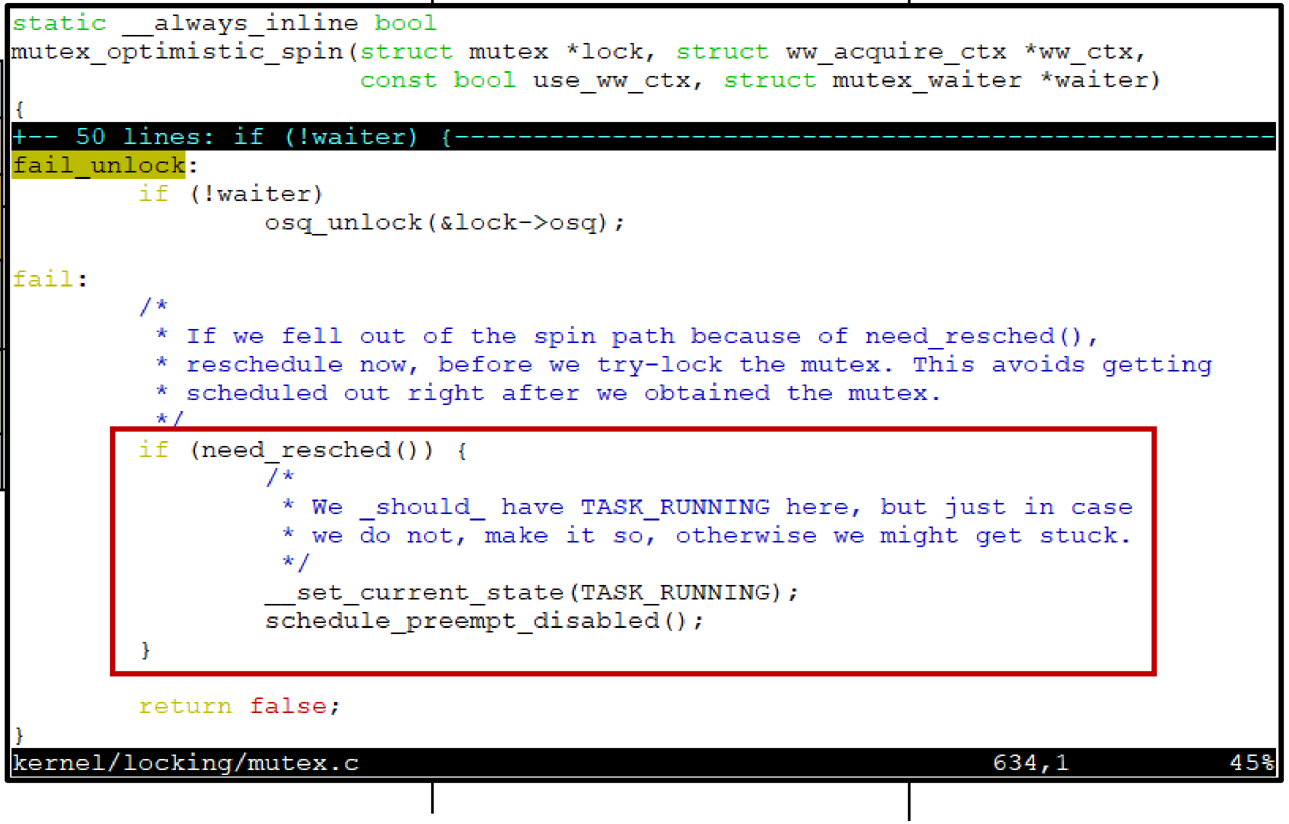


# Three cases for “cannot spin on mutex owner”

- The lock owner is preempted
- The spinning task is preempted
- The lock owner sleeps

# midpath: [Case #3] Spinner (osq lock owner) is preempted



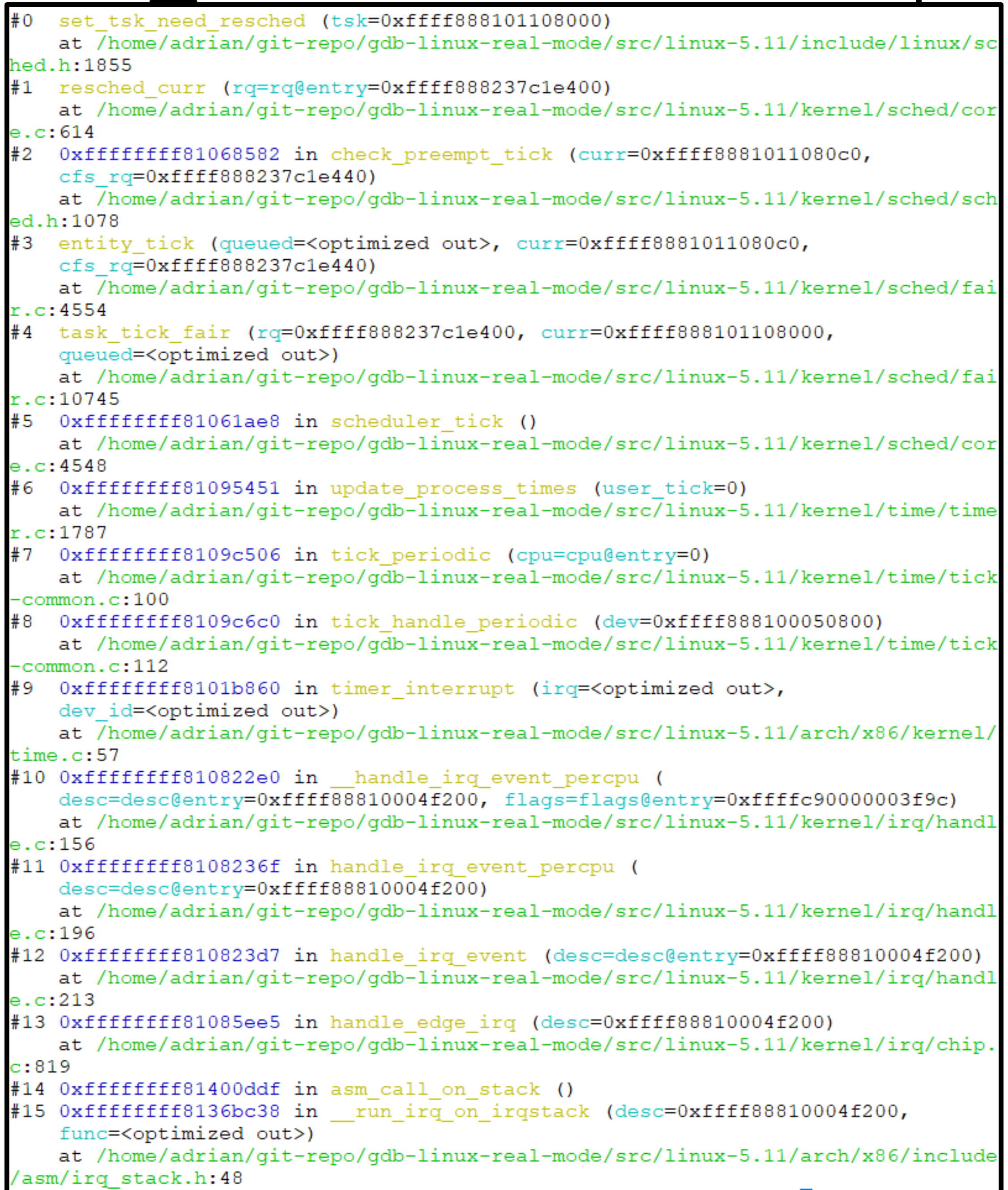


```
static __always_inline bool
mutex_optimistic_spin(struct mutex *lock, struct ww_acquire_ctx *ww_ctx,
                      const bool use_ww_ctx, struct mutex_waiter *waiter)
{
+-- 50 lines: if (!waiter) {---------------------------------------------------
fail_unlock:
        if (!waiter)
                osq_unlock(&lock->osq);

fail:
        /*
         * If we fell out of the spin path because of need_resched(),
         * reschedule now, before we try-lock the mutex. This avoids getting
         * scheduled out right after we obtained the mutex.
         */
        if (need_resched()) {
                /*
                 * We _should_ have TASK_RUNNING here, but just in case
                 * we do not, make it so, otherwise we might get stuck.
                 */
                __set_current_state(TASK_RUNNING);
                schedule_preempt_disabled();
        }

        return false;
}
kernel/locking/mutex.c                                     634,1          45%
```

**Who sets TIF_NEED_RESCHED? → set_tsk_need_resched()**

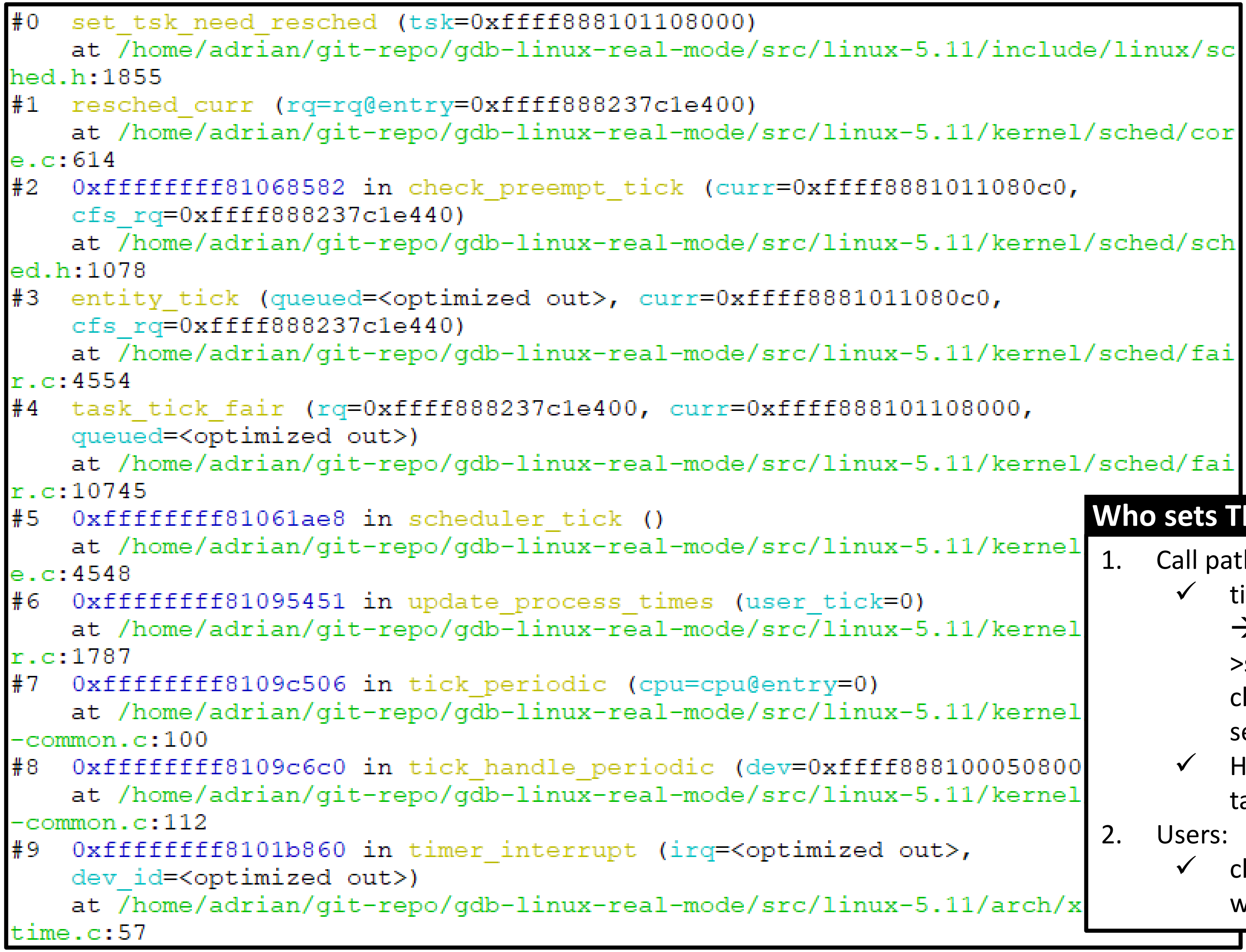
1. Call path
   - ✓ timer_interrupt → tick_handle_periodic → tick_periodic → update_process_times → scheduler_tick → curr->sched_class->task_tick → task_tick_fair → entity_tick -> check_preempt_tick -> **resched_curr** -> set_tsk_need_resched
   - ✓ HW interrupt (not timer HW) → wake up a higher priority task
2. Users:
   - ✓ check_preempt_tick(), check_preempt_wakeup(), wake_up_process()….and so on.

# Who sets TIF_NEED_RESCHED? full call path



```
#0  set_tsk_need_resched (tsk=0xffff888101108000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/include/linux/sc
hed.h:1855
#1  resched_curr (rq=rq@entry=0xffff888237c1e400)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:614
#2  0xffffffff81068582 in check_preempt_tick (curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/sch
ed.h:1078
#3  entity_tick (queued=<optimized out>, curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/fai
r.c:4554
#4  task_tick_fair (rq=0xffff888237c1e400, curr=0xffff888101108000,
    queued=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/fai
r.c:10745
#5  0xffffffff81061ae8 in scheduler_tick ()
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:4548
#6  0xffffffff81095451 in update_process_times (user_tick=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/time/time
r.c:1787
#7  0xffffffff8109c506 in tick_periodic (cpu=cpu@entry=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/time/tick
-common.c:100
#8  0xffffffff8109c6c0 in tick_handle_periodic (dev=0xffff888100050800)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/time/tick
-common.c:112
#9  0xffffffff8101b860 in timer_interrupt (irq=<optimized out>,
    dev_id=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/
time.c:57
#10 0xffffffff810822e0 in __handle_irq_event_percpu (
    desc=desc@entry=0xffff88810004f200, flags=flags@entry=0xffffc90000003f9c)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/irq/handl
e.c:156
#11 0xffffffff8108236f in handle_irq_event_percpu (
    desc=desc@entry=0xffff88810004f200)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/irq/handl
e.c:196
#12 0xffffffff810823d7 in handle_irq_event (desc=desc@entry=0xffff88810004f200)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/irq/handl
e.c:213
#13 0xffffffff81085ee5 in handle_edge_irq (desc=0xffff88810004f200)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/irq/chip.
c:819
#14 0xffffffff81400ddf in asm_call_on_stack ()
#15 0xffffffff8136bc38 in __run_irq_on_irqstack (desc=0xffff88810004f200,
    func=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/include
/asm/irq_stack.h:48
```

# Who sets TIF_NEED_RESCHED?

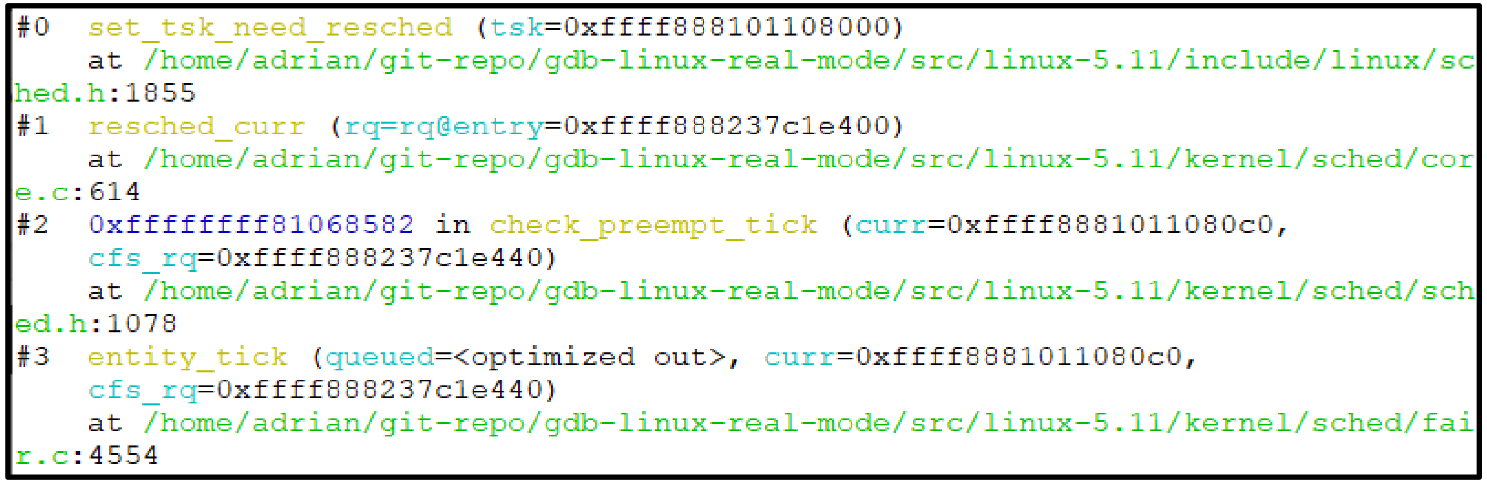


```
#0  set_tsk_need_resched (tsk=0xffff888101108000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/include/linux/sc
hed.h:1855
#1  resched_curr (rq=rq@entry=0xffff888237c1e400)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:614
#2  0xffffffff81068582 in check_preempt_tick (curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/sch
ed.h:1078
#3  entity_tick (queued=<optimized out>, curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/fai
r.c:4554
#4  task_tick_fair (rq=0xffff888237c1e400, curr=0xffff888101108000,
    queued=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/fai
r.c:10745
#5  0xffffffff81061ae8 in scheduler_tick ()
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel
e.c:4548
#6  0xffffffff81095451 in update_process_times (user_tick=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel
r.c:1787
#7  0xffffffff8109c506 in tick_periodic (cpu=cpu@entry=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel
-common.c:100
#8  0xffffffff8109c6c0 in tick_handle_periodic (dev=0xffff888100050800
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel
-common.c:112
#9  0xffffffff8101b860 in timer_interrupt (irq=<optimized out>,
    dev_id=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x
time.c:57
```

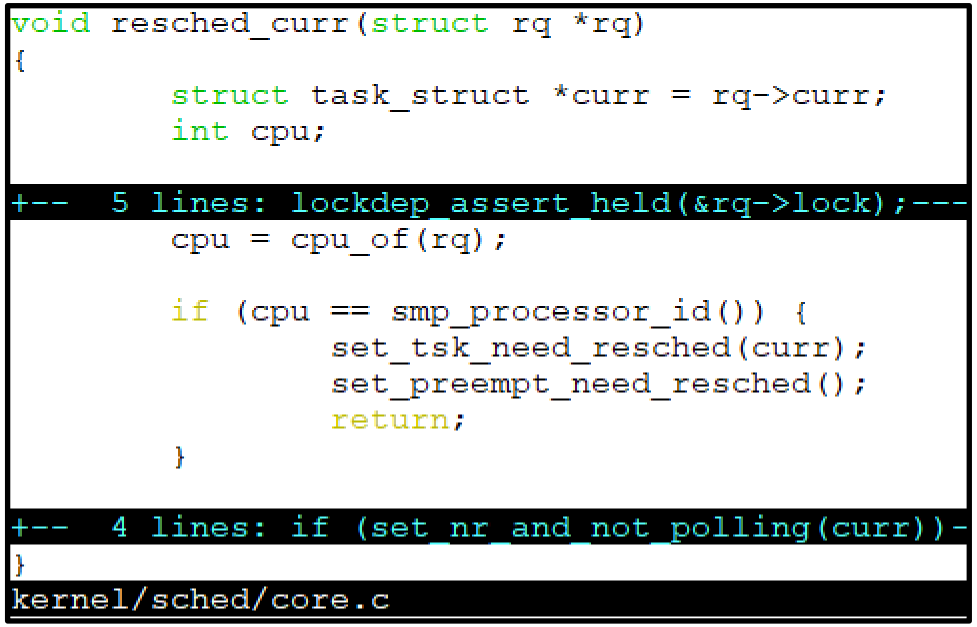
**Who sets TIF_NEED_RESCHED? → set_tsk_need_resched()**

1. Call path
   - ✓ timer_interrupt → tick_handle_periodic → tick_periodic → update_process_times → scheduler_tick → curr->sched_class->task_tick → task_tick_fair → entity_tick -> check_preempt_tick -> **resched_curr** -> set_tsk_need_resched
   - ✓ HW interrupt (not timer HW) → wake up a higher priority task
2. Users:
   - ✓ check_preempt_tick(), check_preempt_wakeup(), wake_up_process()….and so on.

# Who sets TIF_NEED_RESCHED?

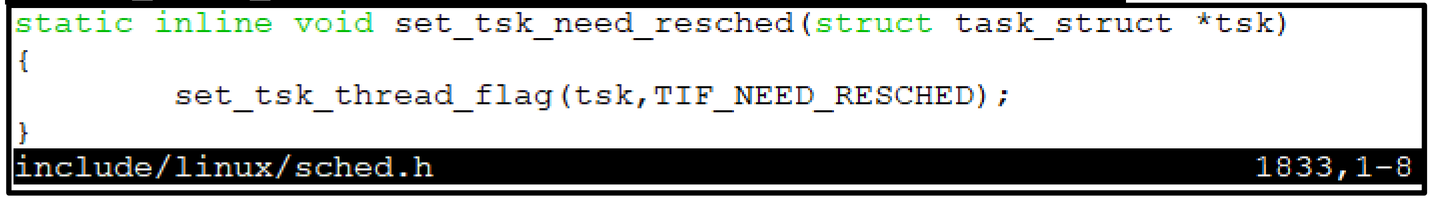


```
#0  set_tsk_need_resched (tsk=0xffff888101108000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/include/linux/sc
hed.h:1855
#1  resched_curr (rq=rq@entry=0xffff888237c1e400)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:614
#2  0xffffffff81068582 in check_preempt_tick (curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/sch
ed.h:1078
#3  entity_tick (queued=<optimized out>, curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/fai
r.c:4554
```



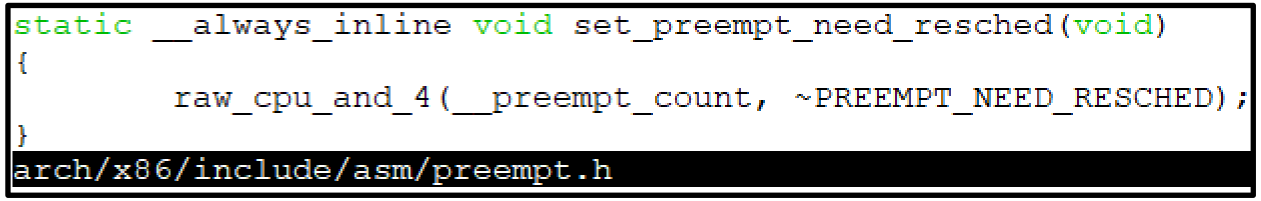
```
void resched_curr(struct rq *rq)
{
        struct task_struct *curr = rq->curr;
        int cpu;

+--  5 lines: lockdep_assert_held(&rq->lock);---
        cpu = cpu_of(rq);

        if (cpu == smp_processor_id()) {
                set_tsk_need_resched(curr);
                set_preempt_need_resched();
                return;
        }

+--  4 lines: if (set_nr_and_not_polling(curr))-
}
kernel/sched/core.c
```

**Set TIF_NEED_RESCHED: current task will be rescheduled later**



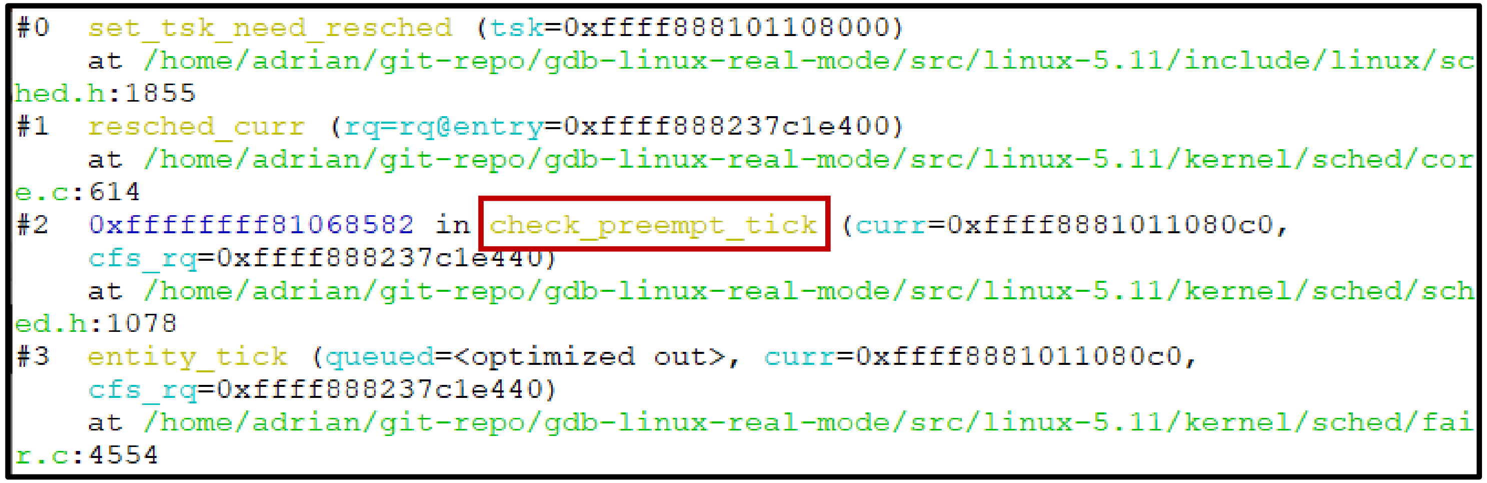
```
static inline void set_tsk_need_resched(struct task_struct *tsk)
{
        set_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
}
include/linux/sched.h                                           1833,1-8
```

**PREEMPT_NEED_RESCHED bit = 0 → Need to reschedule (check comments in this header)**

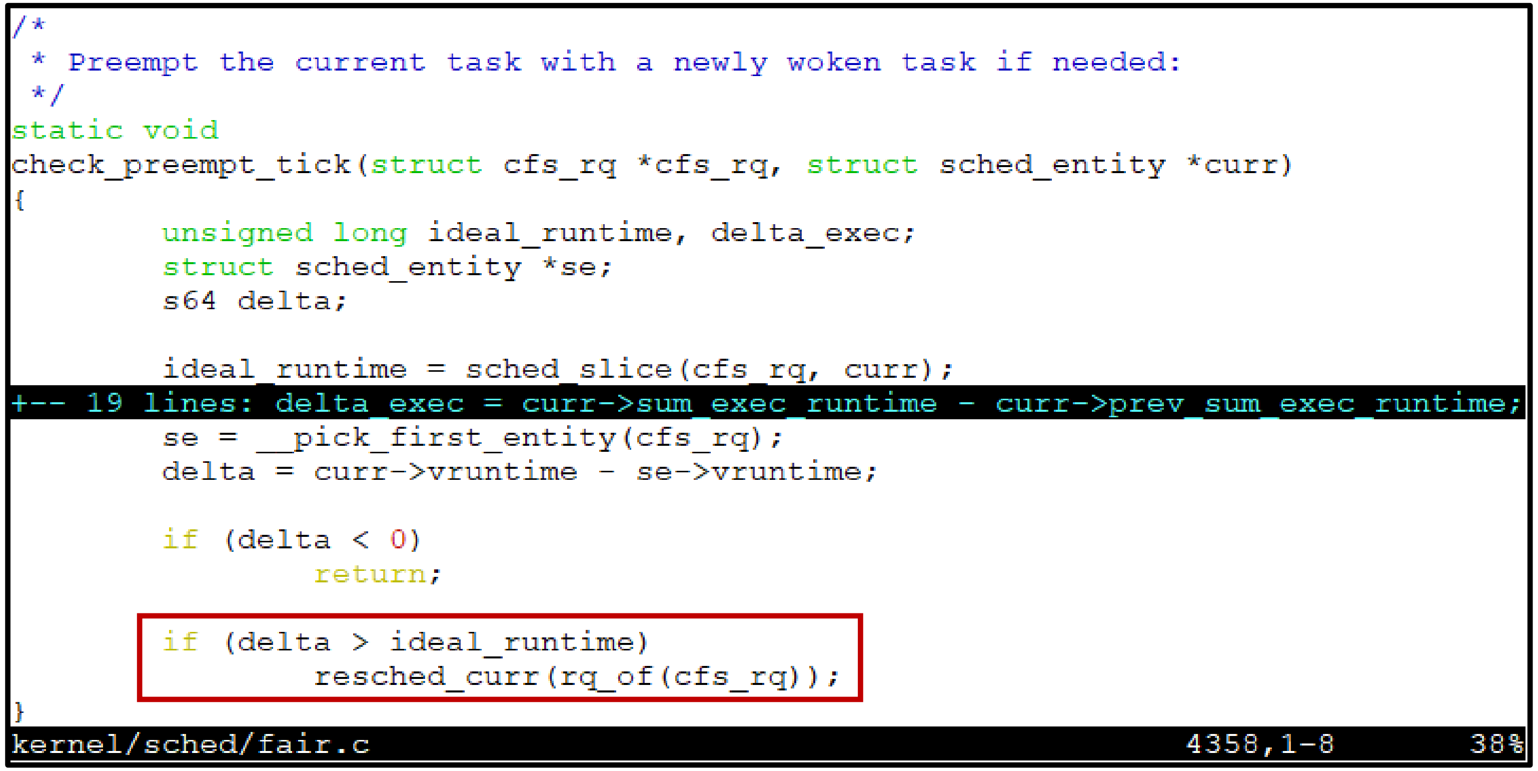


```
static __always_inline void set_preempt_need_resched(void)
{
        raw_cpu_and_4(__preempt_count, ~PREEMPT_NEED_RESCHED);
}
arch/x86/include/asm/preempt.h
```

# Who sets TIF_NEED_RESCHED?



```
#0  set_tsk_need_resched (tsk=0xffff888101108000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/include/linux/sc
hed.h:1855
#1  resched_curr (rq=rq@entry=0xffff888237c1e400)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:614
#2  0xffffffff81068582 in check_preempt_tick (curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/sch
ed.h:1078
#3  entity_tick (queued=<optimized out>, curr=0xffff8881011080c0,
    cfs_rq=0xffff888237c1e440)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/fai
r.c:4554
```



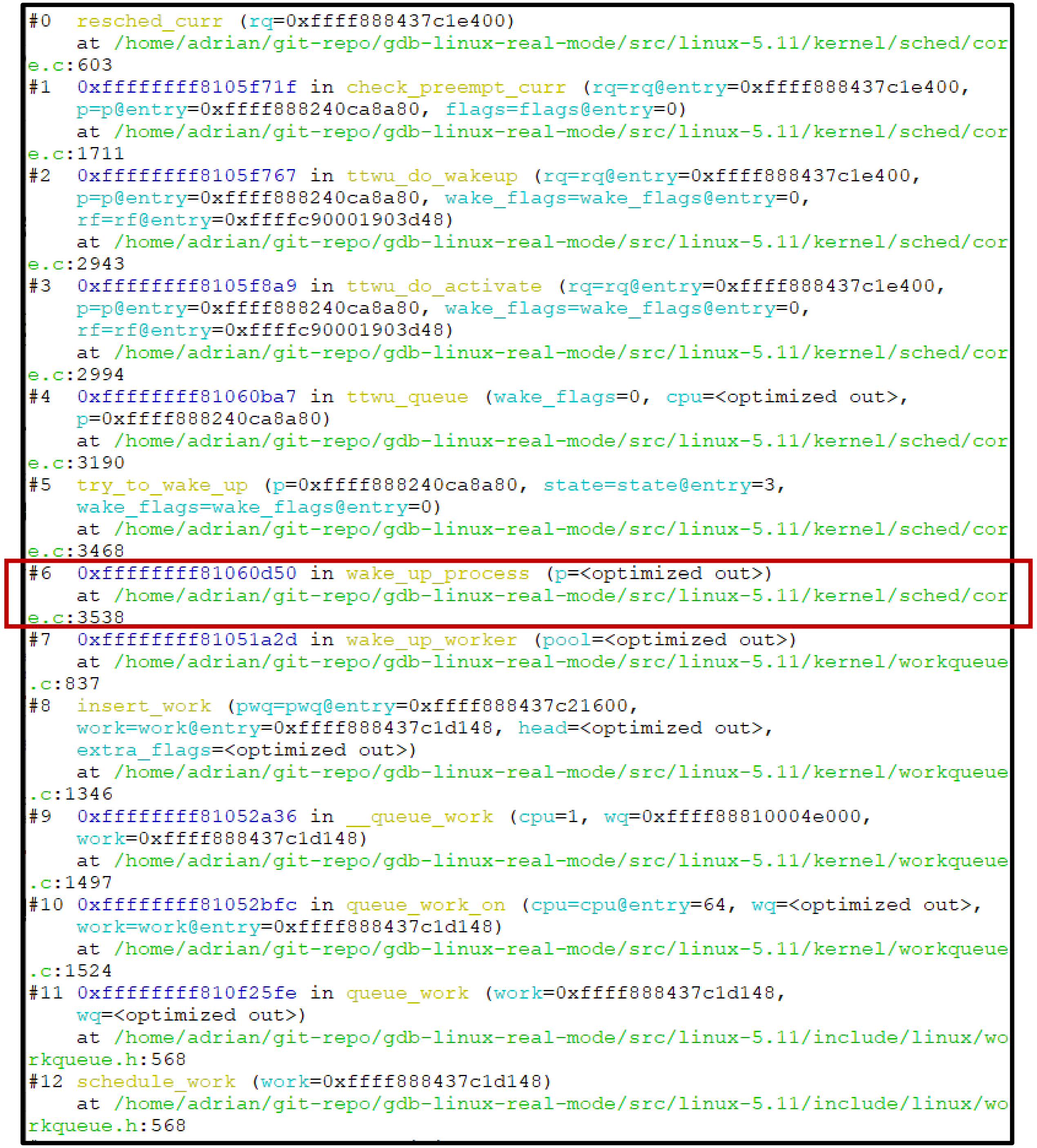
```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        unsigned long ideal_runtime, delta_exec;
        struct sched_entity *se;
        s64 delta;

        ideal_runtime = sched_slice(cfs_rq, curr);
+-- 19 lines: delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
        se = __pick_first_entity(cfs_rq);
        delta = curr->vruntime - se->vruntime;

        if (delta < 0)
                return;

        if (delta > ideal_runtime)
                resched_curr(rq_of(cfs_rq));
}
kernel/sched/fair.c                                             4358,1-8      38%
```

- Set TIF_NEED_RESCHED flag if the delta is greater than ideal_runtime
  - ✓ The running task will be scheduled out.

# Who sets TIF_NEED_RESCHED?

## Who sets TIF_NEED_RESCHED? → set_tsk_need_resched()

1. Call path
   - ✓ timer_interrupt → tick_handle_periodic → tick_periodic → update_process_times → scheduler_tick → curr->sched_class->task_tick → task_tick_fair → entity_tick -> check_preempt_tick -> **resched_curr** -> set_tsk_need_resched
   - ✓ HW interrupt (not timer HW) → wake up a higher priority task
2. Users:
   - ✓ check_preempt_tick(), check_preempt_wakeup(), **wake_up_process()**….and so on.

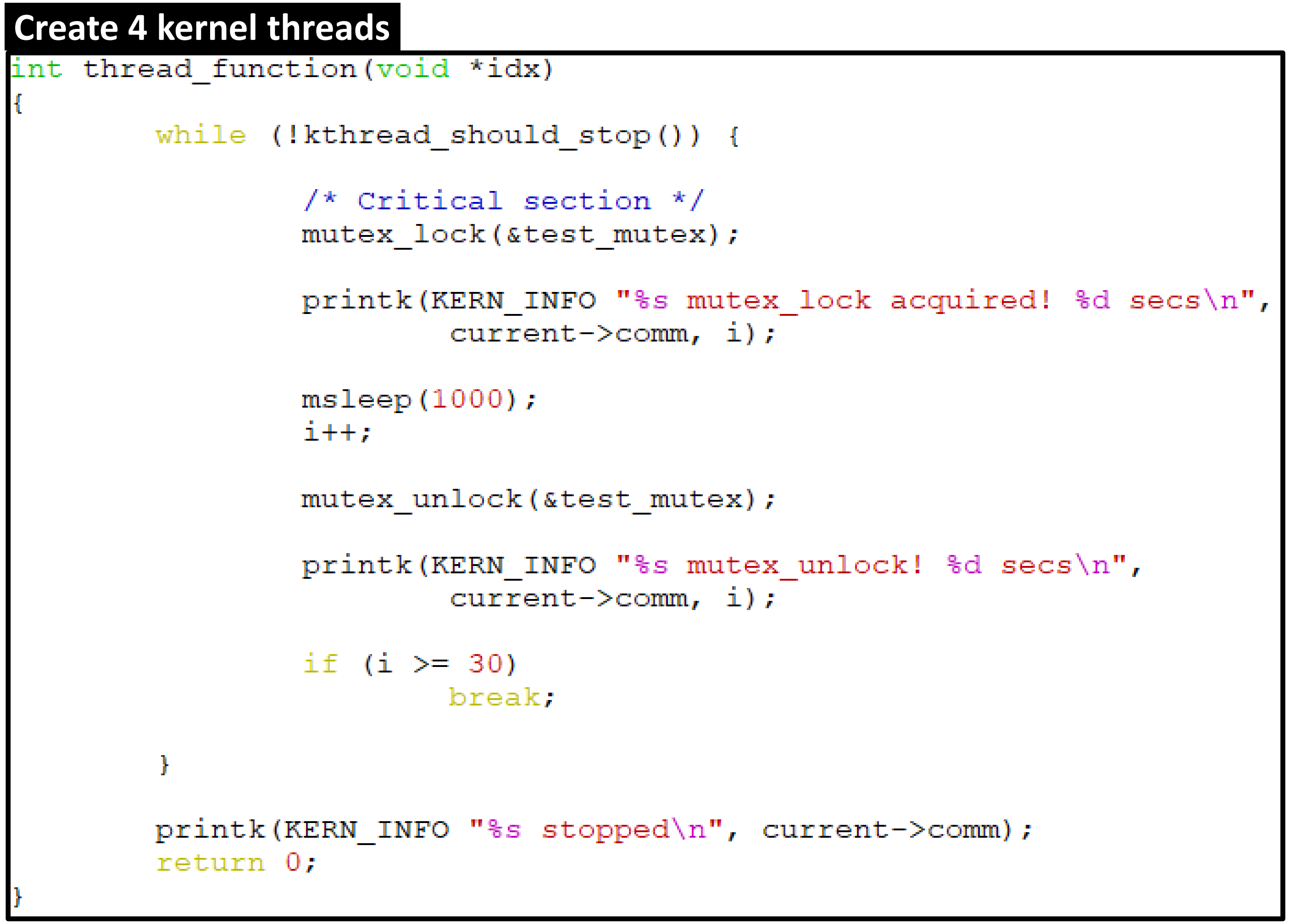


```
#0  resched_curr (rq=0xffff888437c1e400)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:603
#1  0xffffffff8105f71f in check_preempt_curr (rq=rq@entry=0xffff888437c1e400,
    p=p@entry=0xffff888240ca8a80, flags=flags@entry=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:1711
#2  0xffffffff8105f767 in ttwu_do_wakeup (rq=rq@entry=0xffff888437c1e400,
    p=p@entry=0xffff888240ca8a80, wake_flags=wake_flags@entry=0,
    rf=rf@entry=0xffffc90001903d48)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:2943
#3  0xffffffff8105f8a9 in ttwu_do_activate (rq=rq@entry=0xffff888437c1e400,
    p=p@entry=0xffff888240ca8a80, wake_flags=wake_flags@entry=0,
    rf=rf@entry=0xffffc90001903d48)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:2994
#4  0xffffffff81060ba7 in ttwu_queue (wake_flags=0, cpu=<optimized out>,
    p=0xffff888240ca8a80)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:3190
#5  try_to_wake_up (p=0xffff888240ca8a80, state=state@entry=3,
    wake_flags=wake_flags@entry=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:3468
#6  0xffffffff81060d50 in wake_up_process (p=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:3538
#7  0xffffffff81051a2d in wake_up_worker (pool=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/workqueue
.c:837
#8  insert_work (pwq=pwq@entry=0xffff888437c21600,
    work=work@entry=0xffff888437c1d148, head=<optimized out>,
    extra_flags=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/workqueue
.c:1346
#9  0xffffffff81052a36 in __queue_work (cpu=1, wq=0xffff88810004e000,
    work=0xffff888437c1d148)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/workqueue
.c:1497
#10 0xffffffff81052bfc in queue_work_on (cpu=cpu@entry=64, wq=<optimized out>,
    work=work@entry=0xffff888437c1d148)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/workqueue
.c:1524
#11 0xffffffff810f25fe in queue_work (work=0xffff888437c1d148,
    wq=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/include/linux/wo
rkqueue.h:568
#12 schedule_work (work=0xffff888437c1d148)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/include/linux/wo
rkqueue.h:568
```

# Three cases for “cannot spin on mutex owner”

- The lock owner is preempted
- The spinning task is preempted
- The lock owner sleeps

# [Case #4] Locker owner sleeps (reschedule): A test kernel module



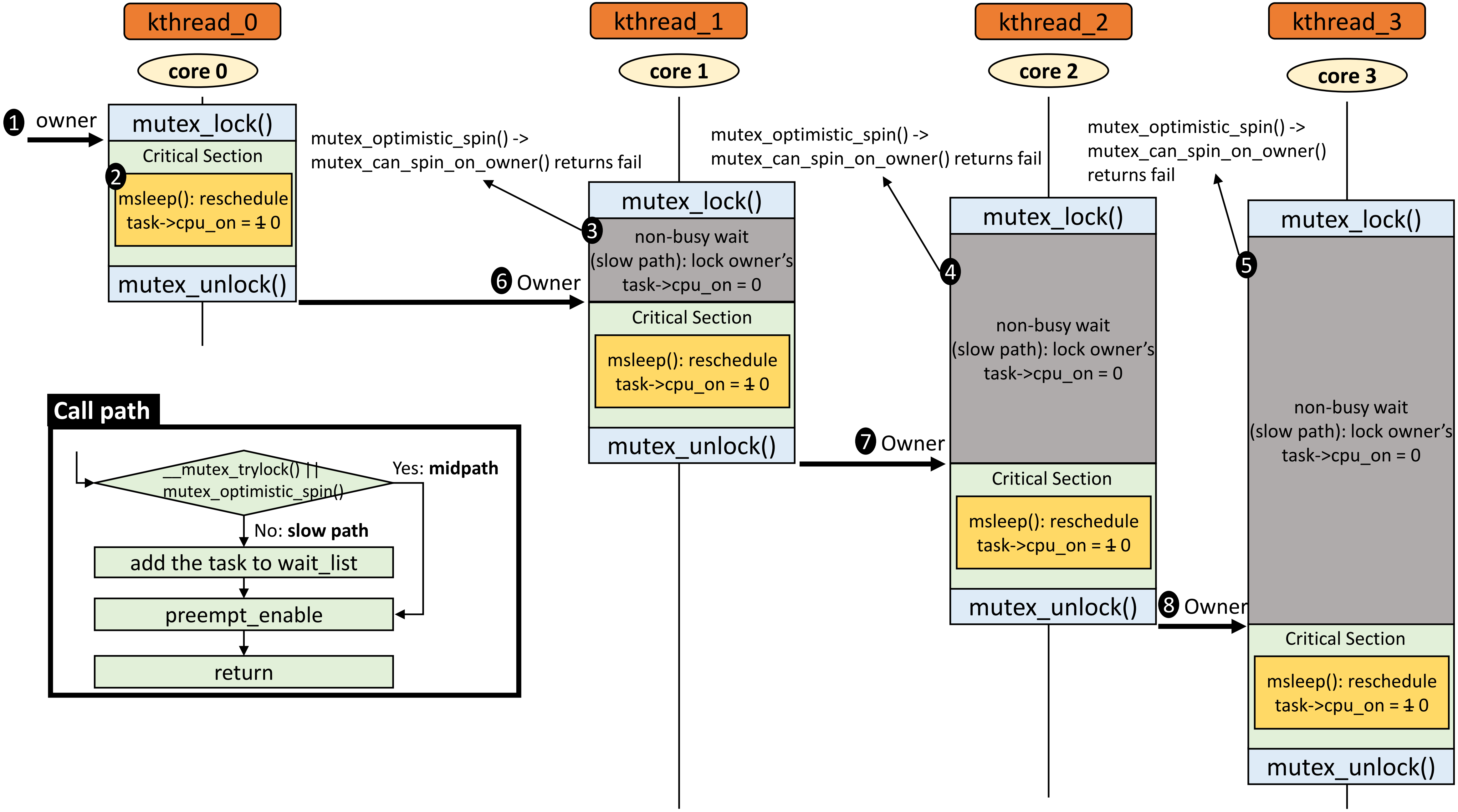
**Create 4 kernel threads**

```
int thread_function(void *idx)
{
        while (!kthread_should_stop()) {

                /* Critical section */
                mutex_lock(&test_mutex);

                printk(KERN_INFO "%s mutex_lock acquired! %d secs\n",
                        current->comm, i);

                msleep(1000);
                i++;

                mutex_unlock(&test_mutex);

                printk(KERN_INFO "%s mutex_unlock! %d secs\n",
                        current->comm, i);

                if (i >= 30)
                        break;

        }

        printk(KERN_INFO "%s stopped\n", current->comm);
        return 0;
}
```

Source code (github): test-modules/mutex/mutex.c

**The action of sleep is identical to preemption and “wait for IO”: reschedule**

# [Case #4] Locker owner sleeps (reschedule): other tasks cannot spin



kthread_0

core 0

1 owner

mutex_lock()

Critical Section

2 msleep(): reschedule task->cpu_on = ~~1~~ 0

mutex_unlock()

6 Owner

kthread_1

core 1

mutex_optimistic_spin() -> mutex_can_spin_on_owner() returns fail

mutex_lock()

3 non-busy wait (slow path): lock owner's task->cpu_on = 0

Critical Section

msleep(): reschedule task->cpu_on = ~~1~~ 0

mutex_unlock()

7 Owner

kthread_2

core 2

mutex_optimistic_spin() -> mutex_can_spin_on_owner() returns fail

mutex_lock()

4 non-busy wait (slow path): lock owner's task->cpu_on = 0

Critical Section

msleep(): reschedule task->cpu_on = ~~1~~ 0

mutex_unlock()

8 Owner

kthread_3

core 3

mutex_optimistic_spin() -> mutex_can_spin_on_owner() returns fail

mutex_lock()

5 non-busy wait (slow path): lock owner's task->cpu_on = 0

Critical Section

msleep(): reschedule task->cpu_on = ~~1~~ 0

mutex_unlock()

**Call path**

__mutex_trylock() || mutex_optimistic_spin()

Yes: **midpath**

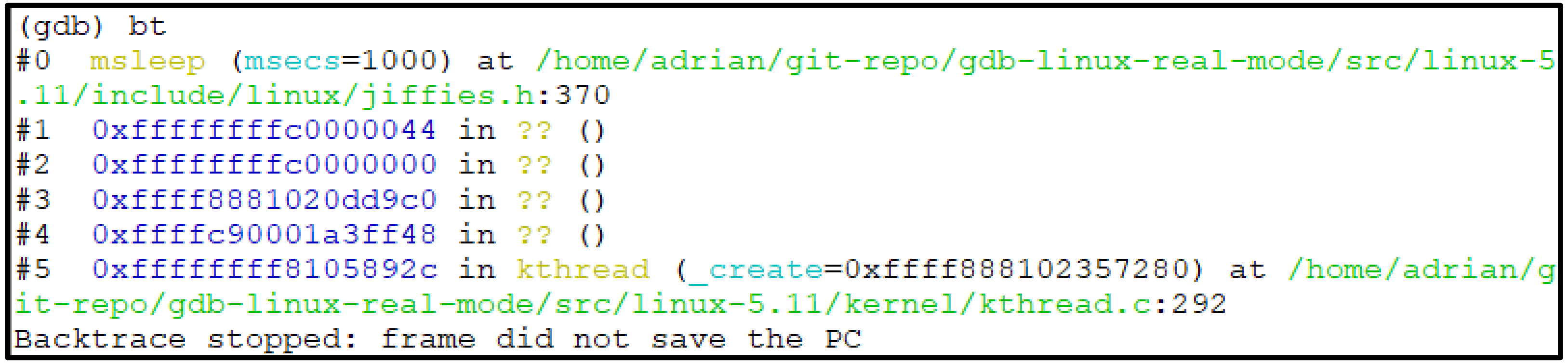
No: **slow path**

add the task to wait_list

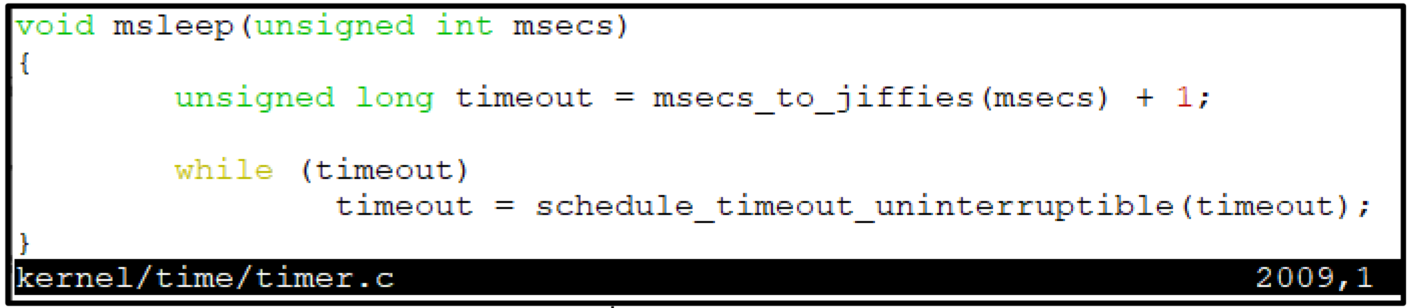
preempt_enable

return

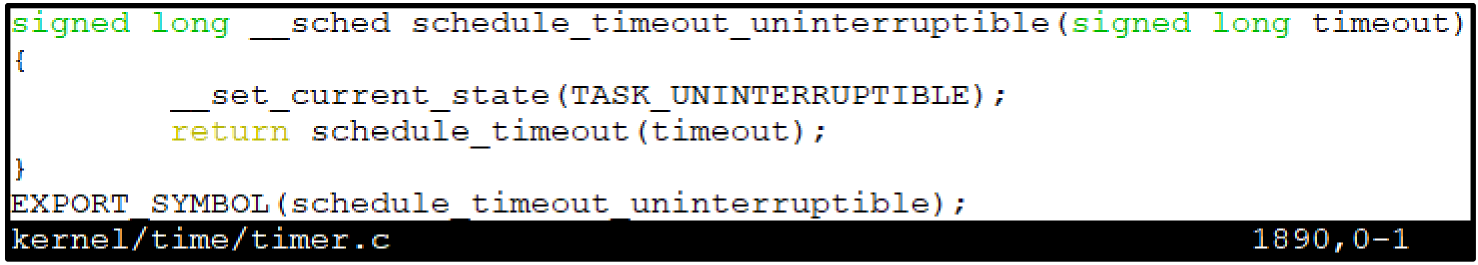
# [Case #4] Locker owner sleeps (reschedule): gdb



```
(gdb) bt
#0  msleep (msecs=1000) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5
.11/include/linux/jiffies.h:370
#1  0xffffffffc0000044 in ?? ()
#2  0xffffffffc0000000 in ?? ()
#3  0xffff8881020dd9c0 in ?? ()
#4  0xffffc90001a3ff48 in ?? ()
#5  0xffffffff8105892c in kthread (_create=0xffff888102357280) at /home/adrian/g
it-repo/gdb-linux-real-mode/src/linux-5.11/kernel/kthread.c:292
Backtrace stopped: frame did not save the PC
```

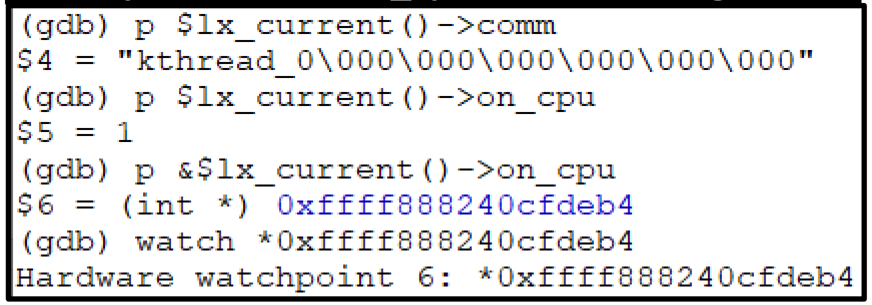


```
void msleep(unsigned int msecs)
{
        unsigned long timeout = msecs_to_jiffies(msecs) + 1;

        while (timeout)
                timeout = schedule_timeout_uninterruptible(timeout);
}
kernel/time/timer.c                                          2009,1
```

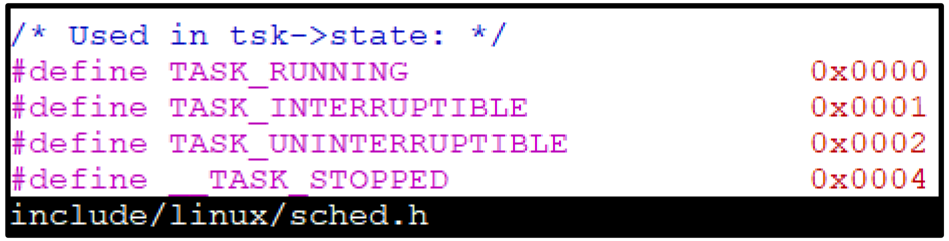


```
signed long __sched schedule_timeout_uninterruptible(signed long timeout)
{
        __set_current_state(TASK_UNINTERRUPTIBLE);
        return schedule_timeout(timeout);
}
EXPORT_SYMBOL(schedule_timeout_uninterruptible);
kernel/time/timer.c                                          1890,0-1
```

**watchpoint: task->on_cpu → who changes this?**

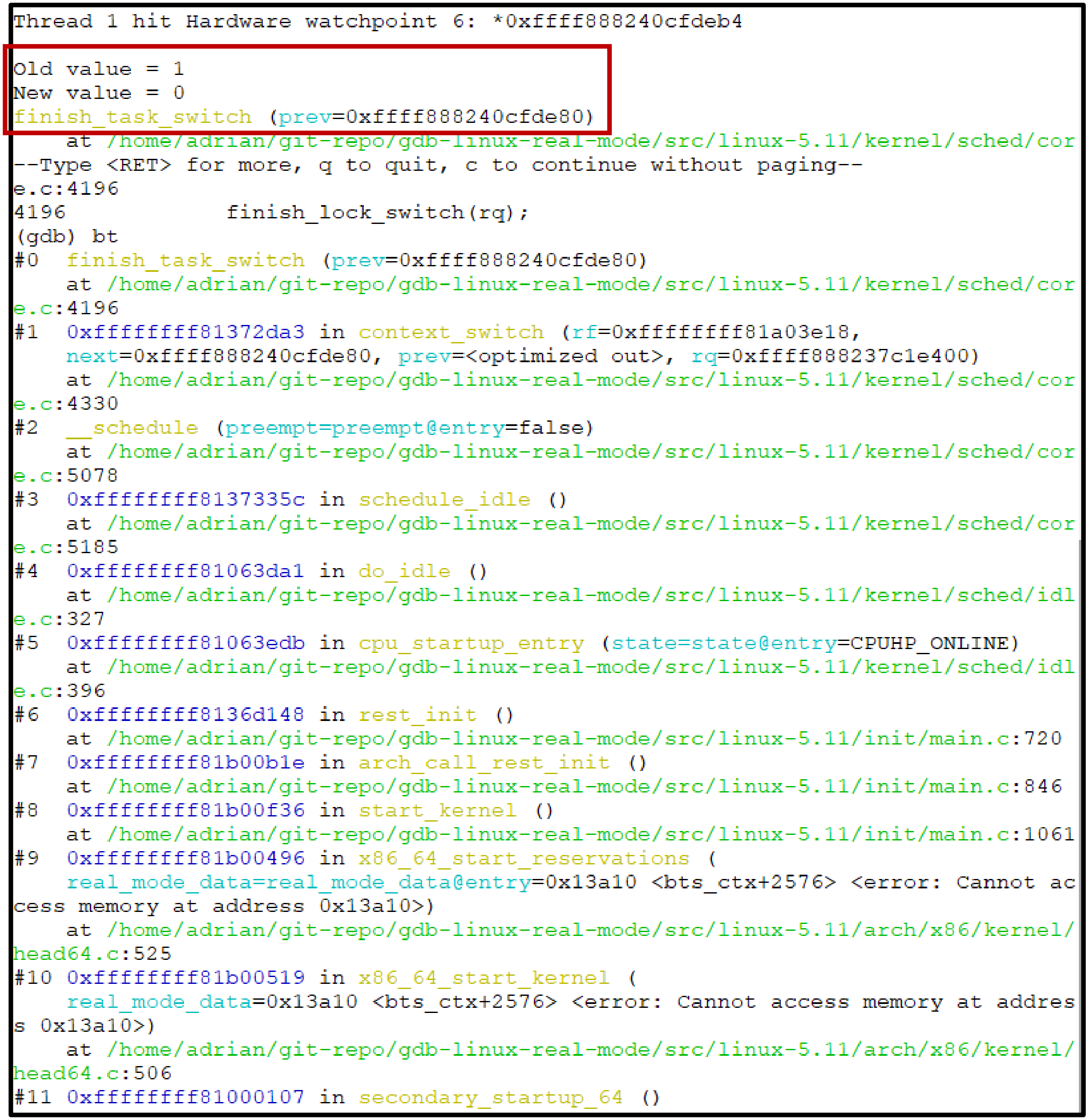


```
(gdb) p $lx_current()->comm
$4 = "kthread_0\000\000\000\000\000\000"
(gdb) p $lx_current()->on_cpu
$5 = 1
(gdb) p &$lx_current()->on_cpu
$6 = (int *) 0xffff888240cfdeb4
(gdb) watch *0xffff888240cfdeb4
Hardware watchpoint 6: *0xffff888240cfdeb4
```

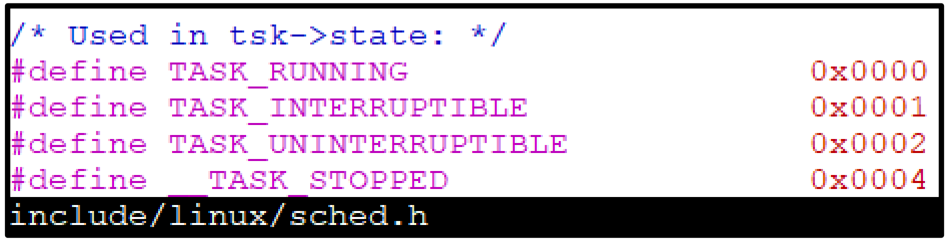


```
/* Used in tsk->state: */
#define TASK_RUNNING                    0x0000
#define TASK_INTERRUPTIBLE              0x0001
#define TASK_UNINTERRUPTIBLE            0x0002
#define __TASK_STOPPED                  0x0004
include/linux/sched.h
```

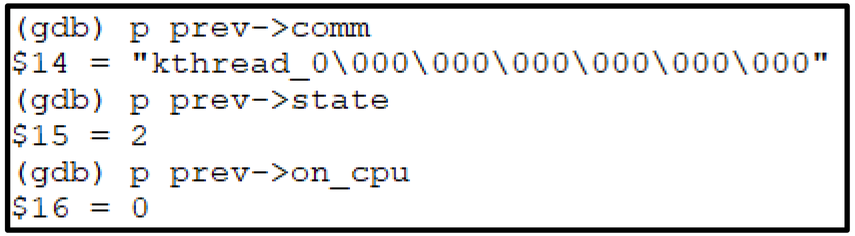
# [Case #4] Locker owner sleeps (reschedule): Who changes task->on_cpu?



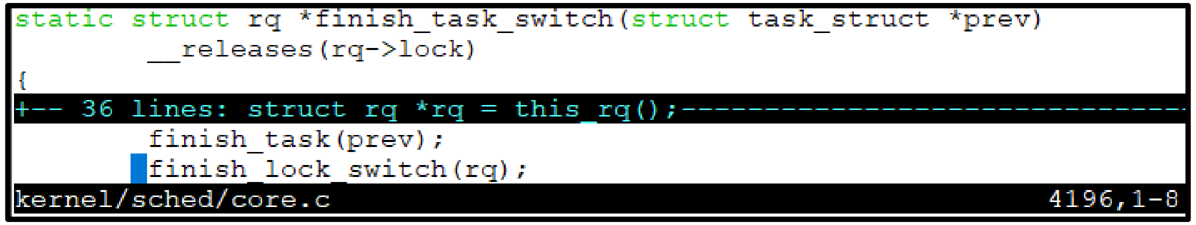
```
Thread 1 hit Hardware watchpoint 6: *0xffff888240cfdeb4

Old value = 1
New value = 0
finish_task_switch (prev=0xffff888240cfde80)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
--Type <RET> for more, q to quit, c to continue without paging--
e.c:4196
4196            finish_lock_switch(rq);
(gdb) bt
#0  finish_task_switch (prev=0xffff888240cfde80)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:4196
#1  0xffffffff81372da3 in context_switch (rf=0xffffffff81a03e18,
    next=0xffff888240cfde80, prev=<optimized out>, rq=0xffff888237c1e400)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:4330
#2  __schedule (preempt=preempt@entry=false)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:5078
#3  0xffffffff8137335c in schedule_idle ()
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:5185
#4  0xffffffff81063da1 in do_idle ()
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/idl
e.c:327
#5  0xffffffff81063edb in cpu_startup_entry (state=state@entry=CPUHP_ONLINE)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/idl
e.c:396
#6  0xffffffff8136d148 in rest_init ()
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/init/main.c:720
#7  0xffffffff81b00b1e in arch_call_rest_init ()
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/init/main.c:846
#8  0xffffffff81b00f36 in start_kernel ()
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/init/main.c:1061
#9  0xffffffff81b00496 in x86_64_start_reservations (
    real_mode_data=real_mode_data@entry=0x13a10 <bts_ctx+2576> <error: Cannot ac
cess memory at address 0x13a10>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/
head64.c:525
#10 0xffffffff81b00519 in x86_64_start_kernel (
    real_mode_data=0x13a10 <bts_ctx+2576> <error: Cannot access memory at addres
s 0x13a10>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/kernel/
head64.c:506
#11 0xffffffff81000107 in secondary_startup_64 ()
```



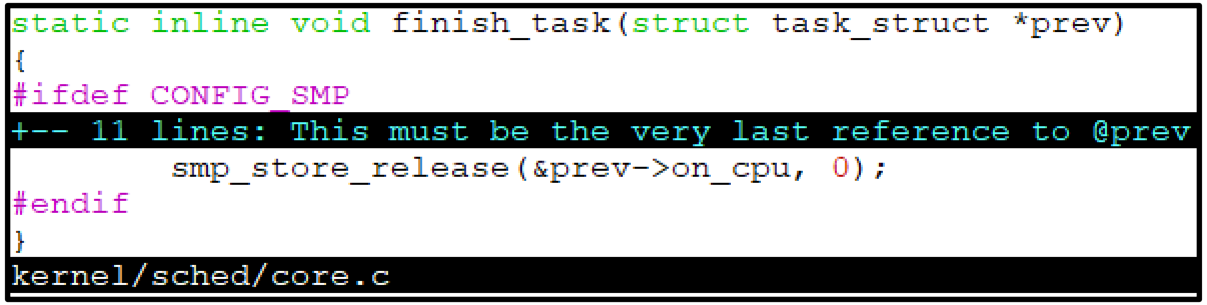
```
/* Used in tsk->state: */
#define TASK_RUNNING                    0x0000
#define TASK_INTERRUPTIBLE              0x0001
#define TASK_UNINTERRUPTIBLE            0x0002
#define __TASK_STOPPED                  0x0004
include/linux/sched.h
```



```
(gdb) p prev->comm
$14 = "kthread_0\000\000\000\000\000\000"
(gdb) p prev->state
$15 = 2
(gdb) p prev->on_cpu
$16 = 0
```



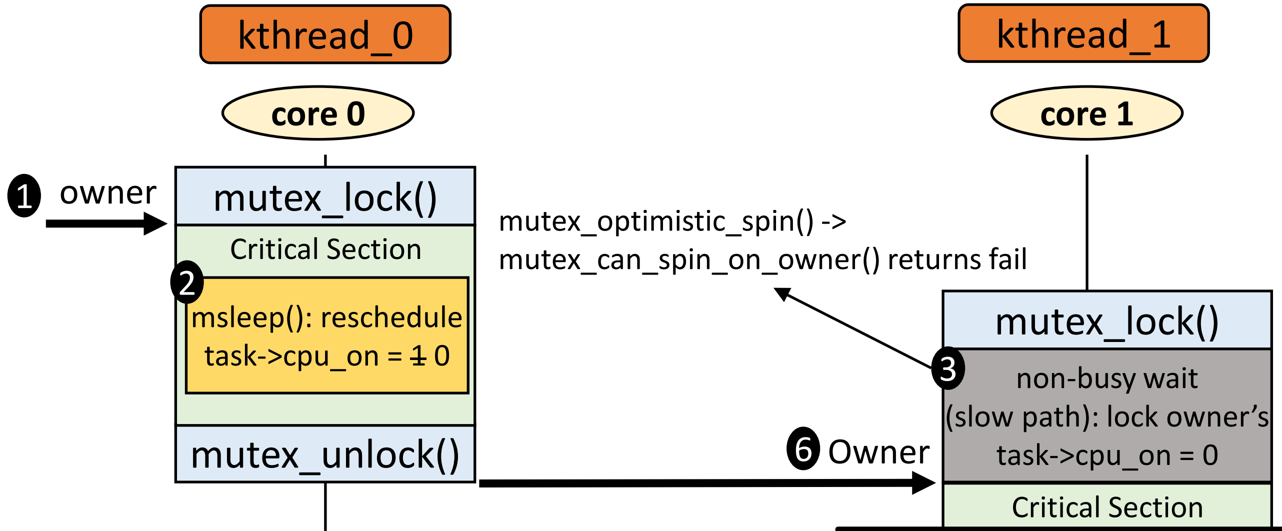
```
static struct rq *finish_task_switch(struct task_struct *prev)
        __releases(rq->lock)
{
+-- 36 lines: struct rq *rq = this_rq();-----------------------------
        finish_task(prev);
        finish_lock_switch(rq);
kernel/sched/core.c                                          4196,1-8
```

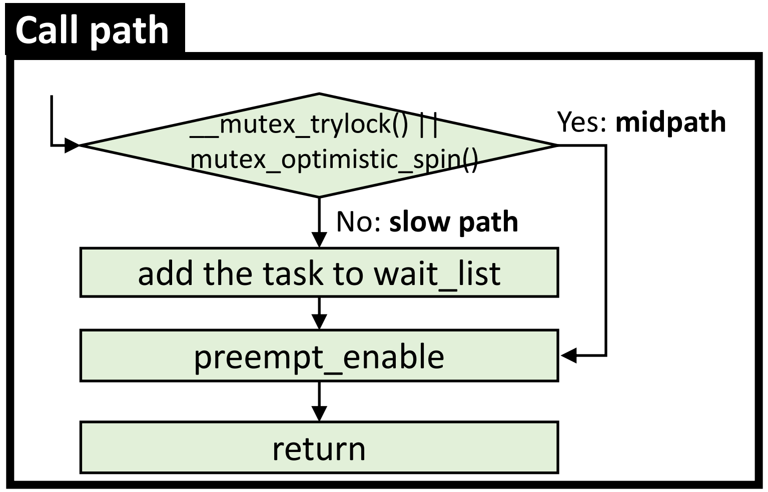


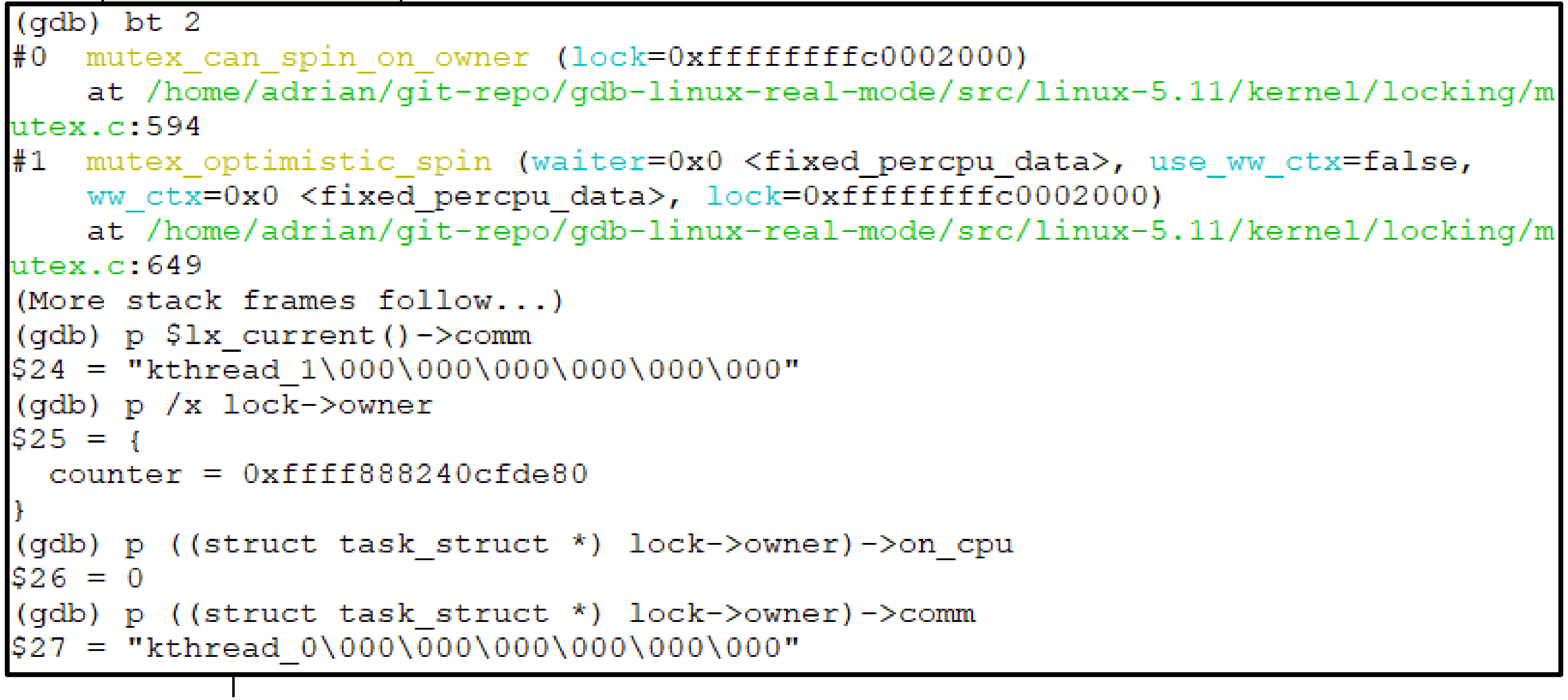
```
static inline void finish_task(struct task_struct *prev)
{
#ifdef CONFIG_SMP
+-- 11 lines: This must be the very last reference to @prev
        smp_store_release(&prev->on_cpu, 0);
#endif
}
kernel/sched/core.c
```

**task->on_cpu is set 0 during context switch**

# [Case #4] Locker owner sleeps (reschedule): gdb: other tasks cannot spin

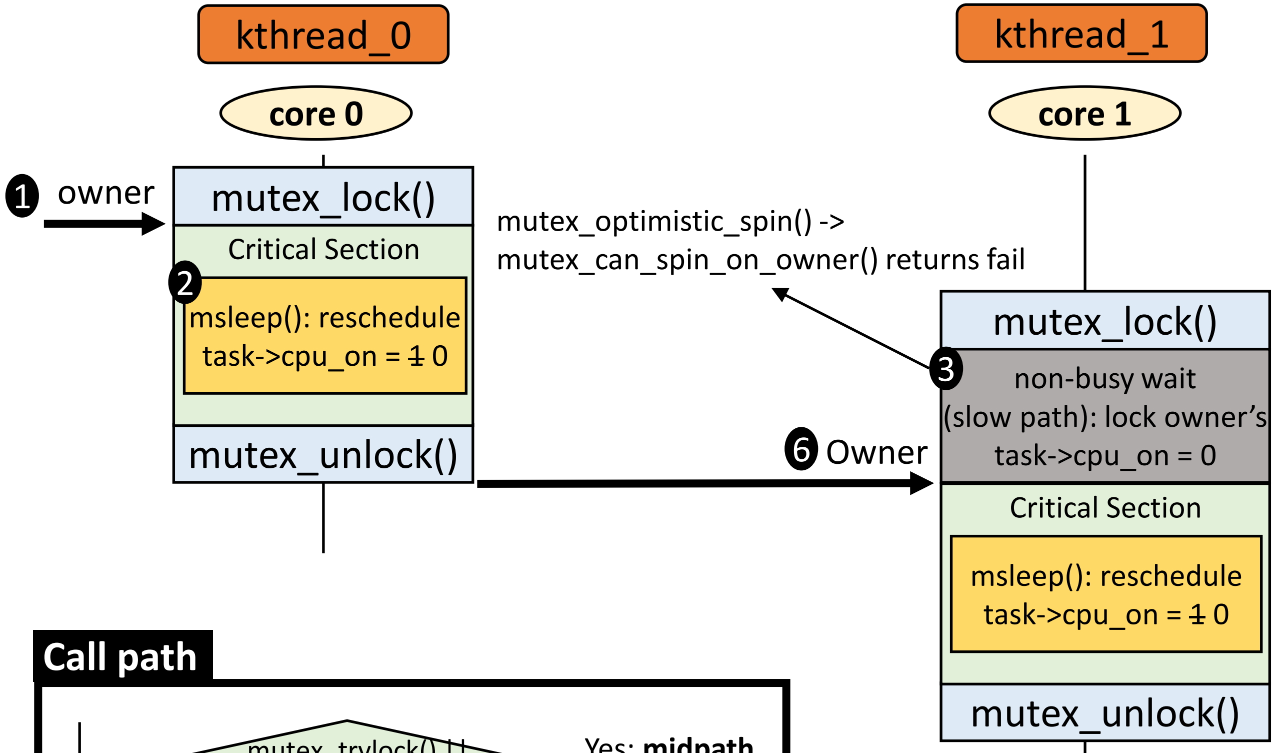


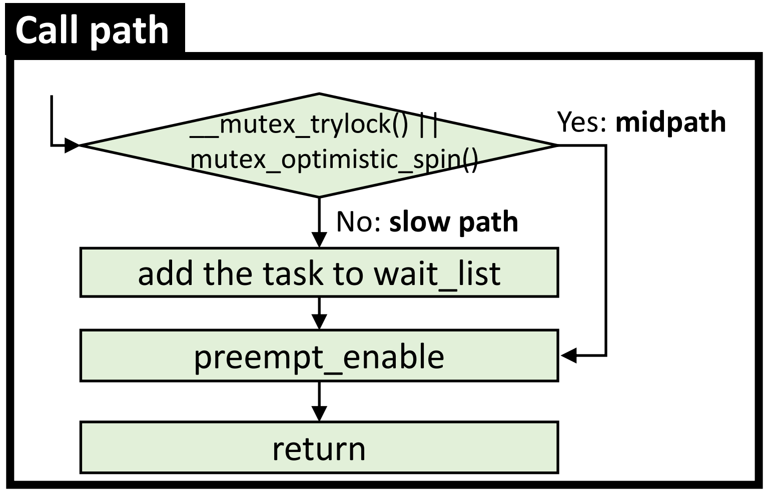


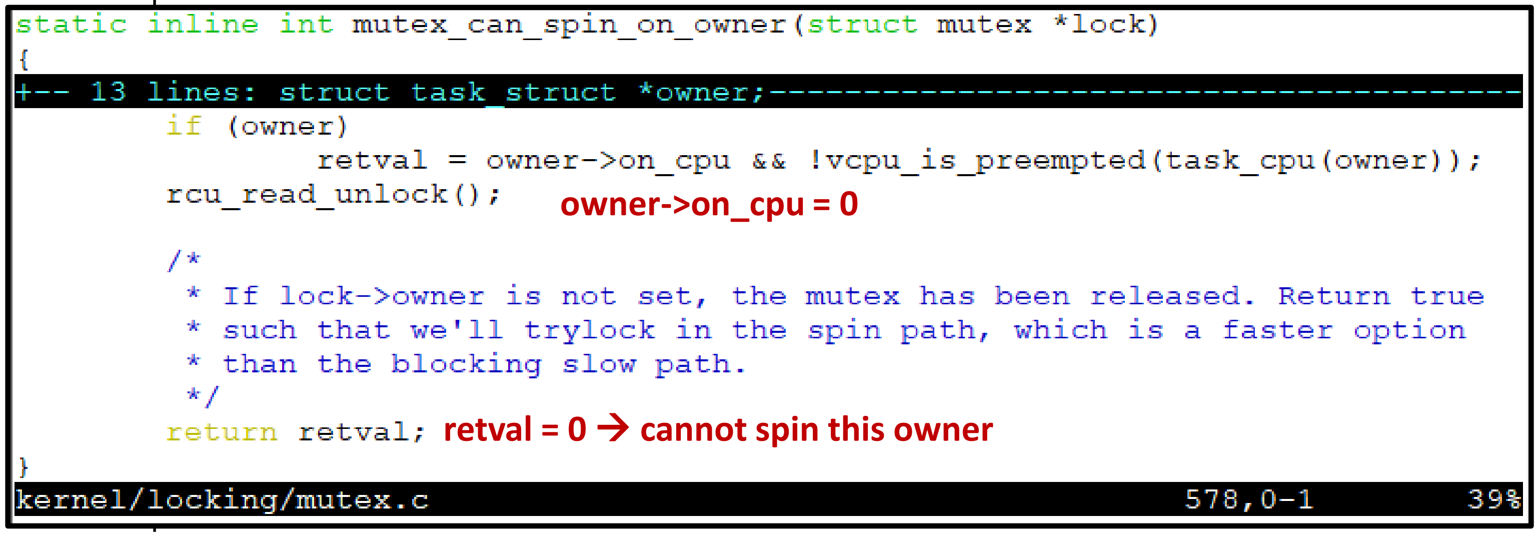


```
(gdb) bt 2
#0  mutex_can_spin_on_owner (lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:594
#1  mutex_optimistic_spin (waiter=0x0 <fixed_percpu_data>, use_ww_ctx=false,
    ww_ctx=0x0 <fixed_percpu_data>, lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:649
(More stack frames follow...)
(gdb) p $lx_current()->comm
$24 = "kthread_1\000\000\000\000\000\000"
(gdb) p /x lock->owner
$25 = {
  counter = 0xffff888240cfde80
}
(gdb) p ((struct task_struct *) lock->owner)->on_cpu
$26 = 0
(gdb) p ((struct task_struct *) lock->owner)->comm
$27 = "kthread_0\000\000\000\000\000\000"
```

# [Case #4] Locker owner sleeps (reschedule): gdb: other tasks cannot spin



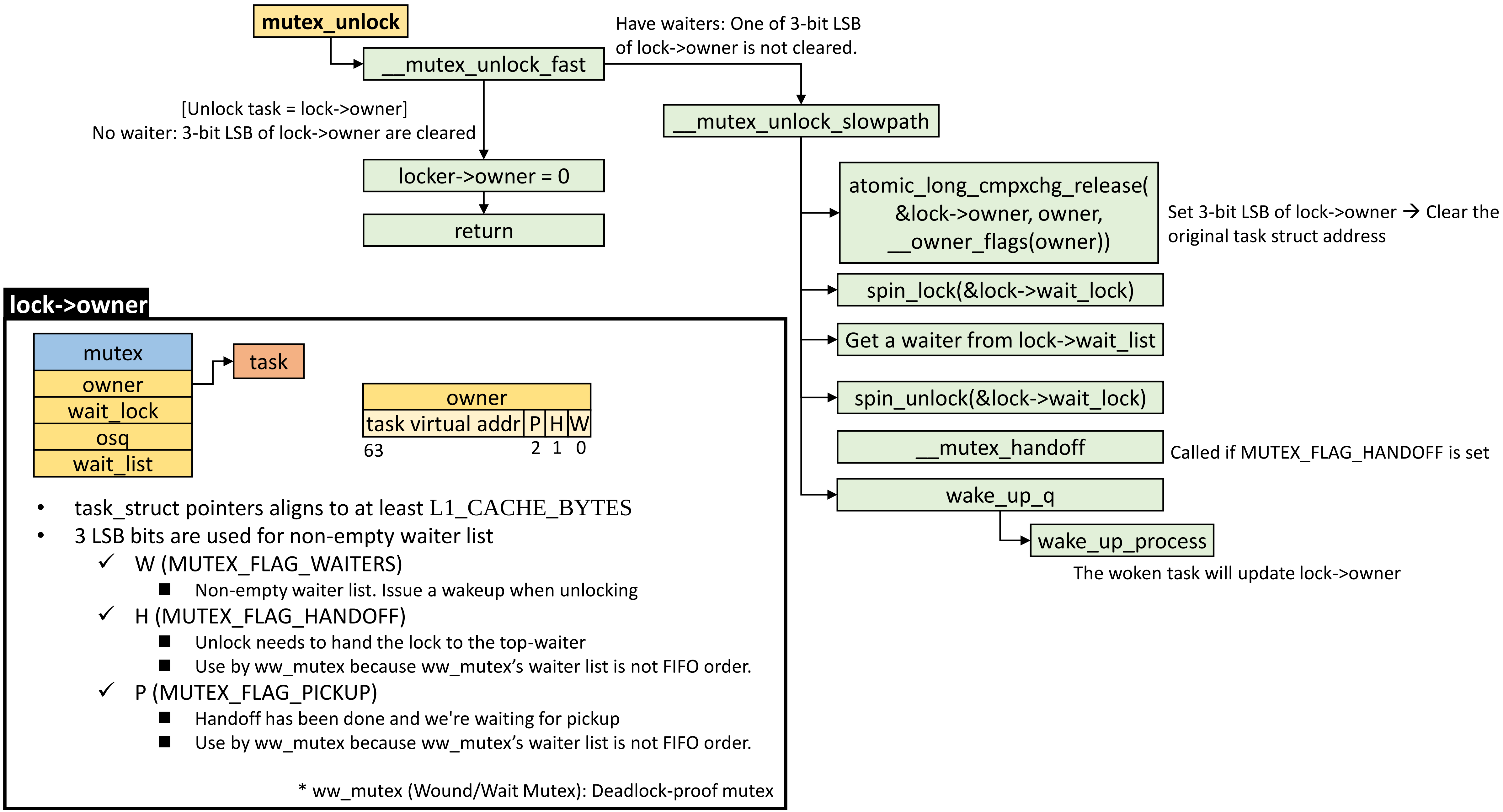




```
static inline int mutex_can_spin_on_owner(struct mutex *lock)
{
+-- 13 lines: struct task_struct *owner;------------------------------------------
        if (owner)
                retval = owner->on_cpu && !vcpu_is_preempted(task_cpu(owner));
        rcu_read_unlock();    owner->on_cpu = 0

        /*
         * If lock->owner is not set, the mutex has been released. Return true
         * such that we'll trylock in the spin path, which is a faster option
         * than the blocking slow path.
         */
        return retval; retval = 0 → cannot spin this owner
}
kernel/locking/mutex.c                                          578,0-1       39%
```

# mutex_unlock()

# mutex_unlock(): Call path



**mutex_unlock**

__mutex_unlock_fast

[Unlock task = lock->owner]
No waiter: 3-bit LSB of lock->owner are cleared

locker->owner = 0

return

Have waiters: One of 3-bit LSB of lock->owner is not cleared.

__mutex_unlock_slowpath

atomic_long_cmpxchg_release( &lock->owner, owner, __owner_flags(owner))

Set 3-bit LSB of lock->owner → Clear the original task struct address

spin_lock(&lock->wait_lock)

Get a waiter from lock->wait_list

spin_unlock(&lock->wait_lock)

__mutex_handoff

Called if MUTEX_FLAG_HANDOFF is set

wake_up_q

wake_up_process

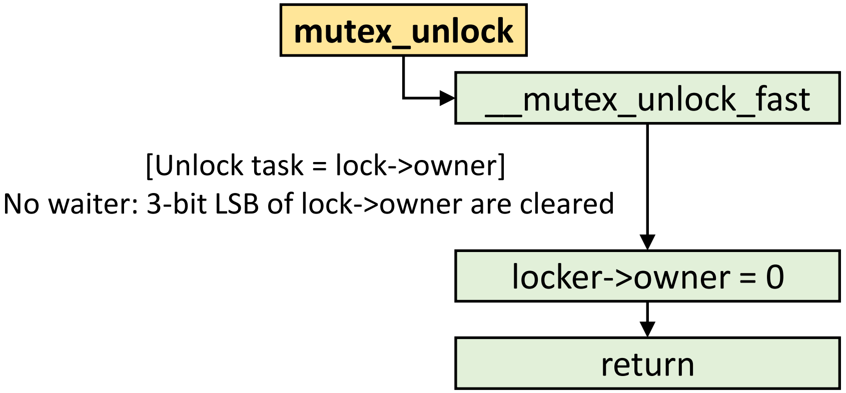
The woken task will update lock->owner

**lock->owner**

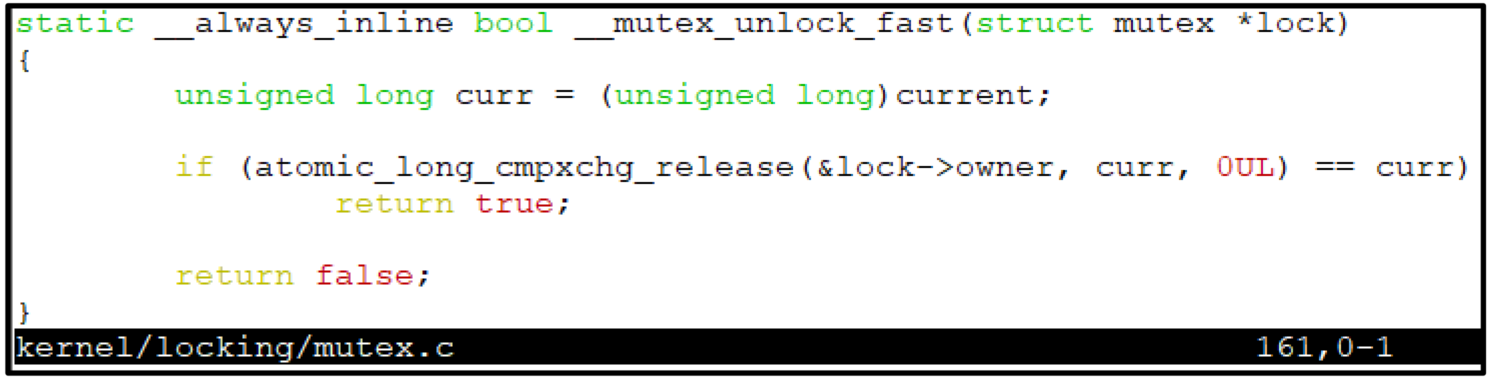
- task_struct pointers aligns to at least L1_CACHE_BYTES
- 3 LSB bits are used for non-empty waiter list
  - ✓ W (MUTEX_FLAG_WAITERS)
    - ■ Non-empty waiter list. Issue a wakeup when unlocking
  - ✓ H (MUTEX_FLAG_HANDOFF)
    - ■ Unlock needs to hand the lock to the top-waiter
    - ■ Use by ww_mutex because ww_mutex’s waiter list is not FIFO order.
  - ✓ P (MUTEX_FLAG_PICKUP)
    - ■ Handoff has been done and we're waiting for pickup
    - ■ Use by ww_mutex because ww_mutex’s waiter list is not FIFO order.

* ww_mutex (Wound/Wait Mutex): Deadlock-proof mutex

# mutex_unlock(): fast path



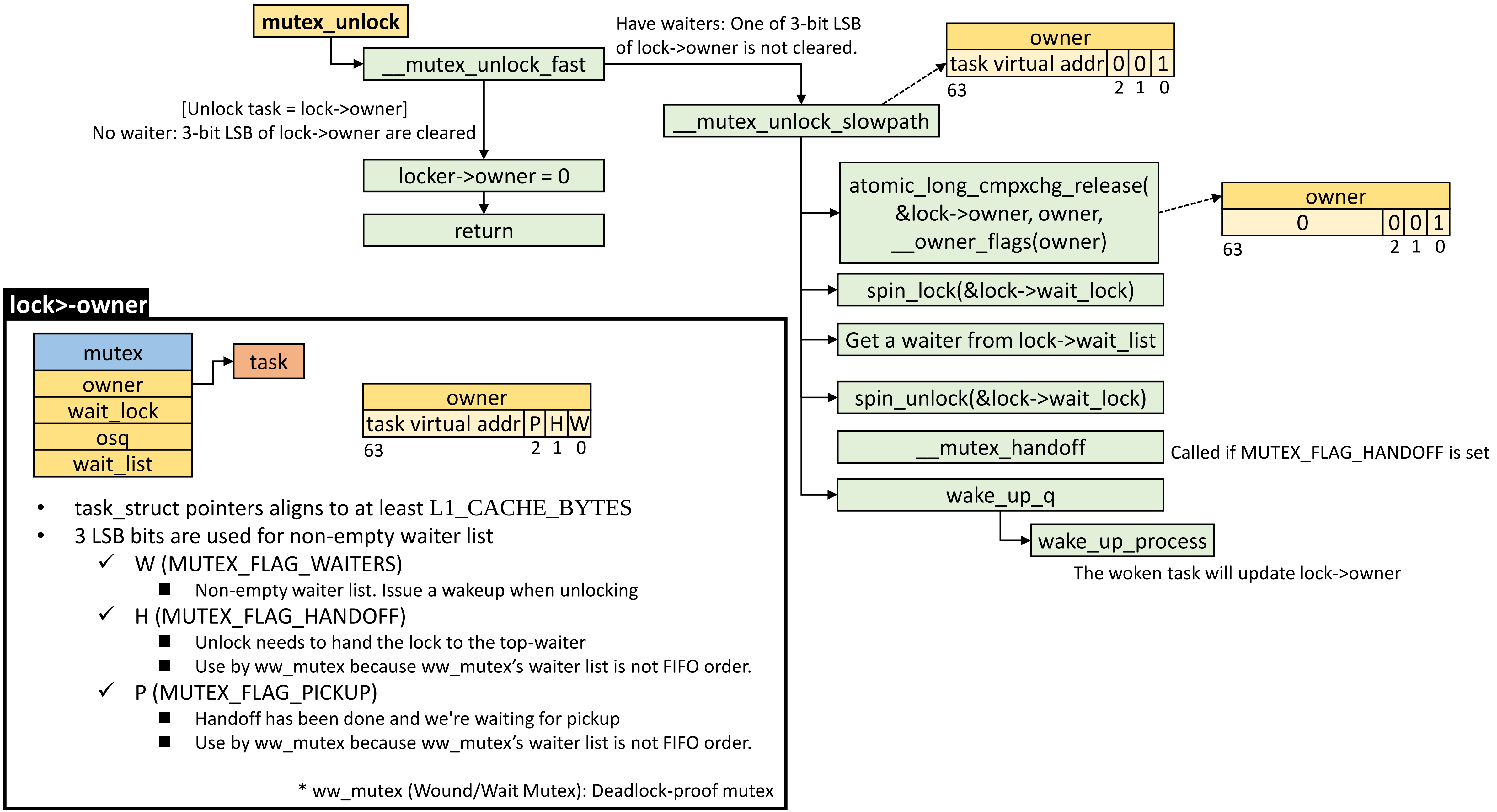
mutex_unlock
__mutex_unlock_fast
[Unlock task = lock->owner]
No waiter: 3-bit LSB of lock->owner are cleared
locker->owner = 0
return



```
static __always_inline bool __mutex_unlock_fast(struct mutex *lock)
{
        unsigned long curr = (unsigned long)current;

        if (atomic_long_cmpxchg_release(&lock->owner, curr, 0UL) == curr)
                return true;

        return false;
}
kernel/locking/mutex.c                                                161,0-1
```

**[fast path] A spinner will take the lock**

# mutex_unlock(): slow path



**mutex_unlock**

__mutex_unlock_fast

[Unlock task = lock->owner]
No waiter: 3-bit LSB of lock->owner are cleared

locker->owner = 0

return

Have waiters: One of 3-bit LSB of lock->owner is not cleared.

__mutex_unlock_slowpath

| owner | | | |
|---|---|---|---|
| task virtual addr | 0 | 0 | 1 |
| 63 | 2 | 1 | 0 |

atomic_long_cmpxchg_release( &lock->owner, owner, __owner_flags(owner))

| owner | | | |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 63 | 2 | 1 | 0 |

spin_lock(&lock->wait_lock)

Get a waiter from lock->wait_list

spin_unlock(&lock->wait_lock)

__mutex_handoff

Called if MUTEX_FLAG_HANDOFF is set

wake_up_q

wake_up_process

The woken task will update lock->owner

**lock>-owner**

mutex

owner

wait_lock

osq

wait_list

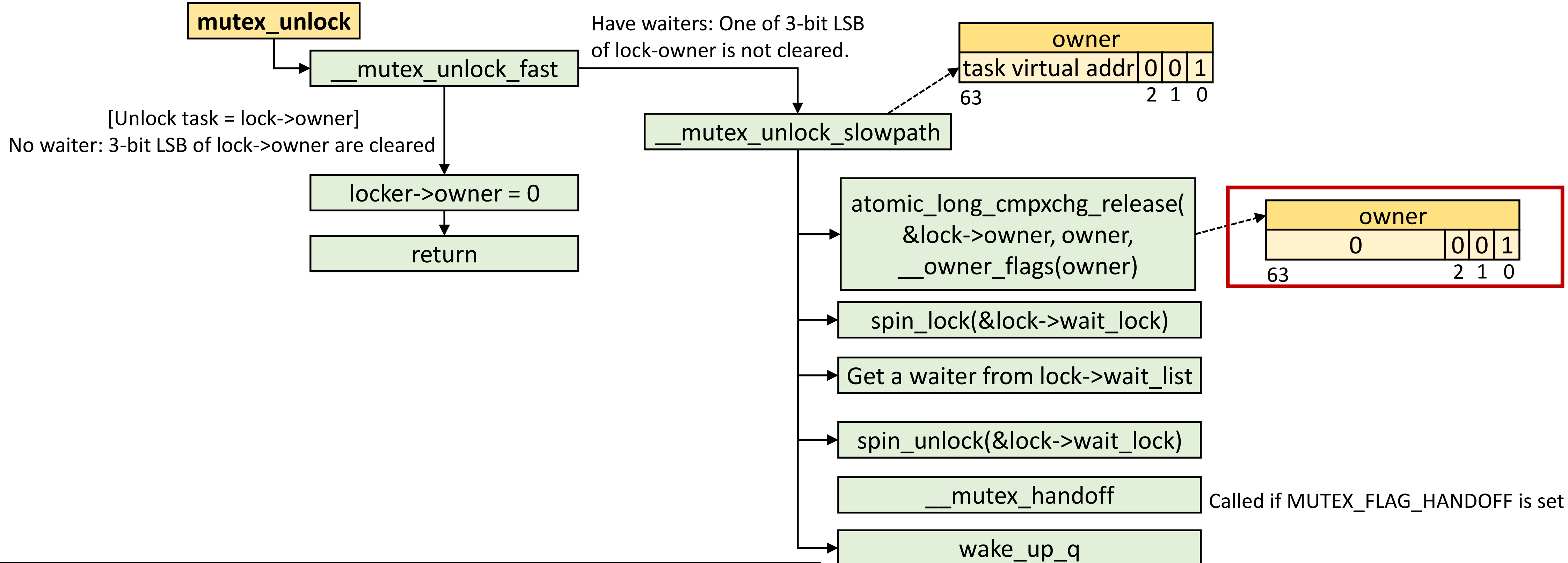
task

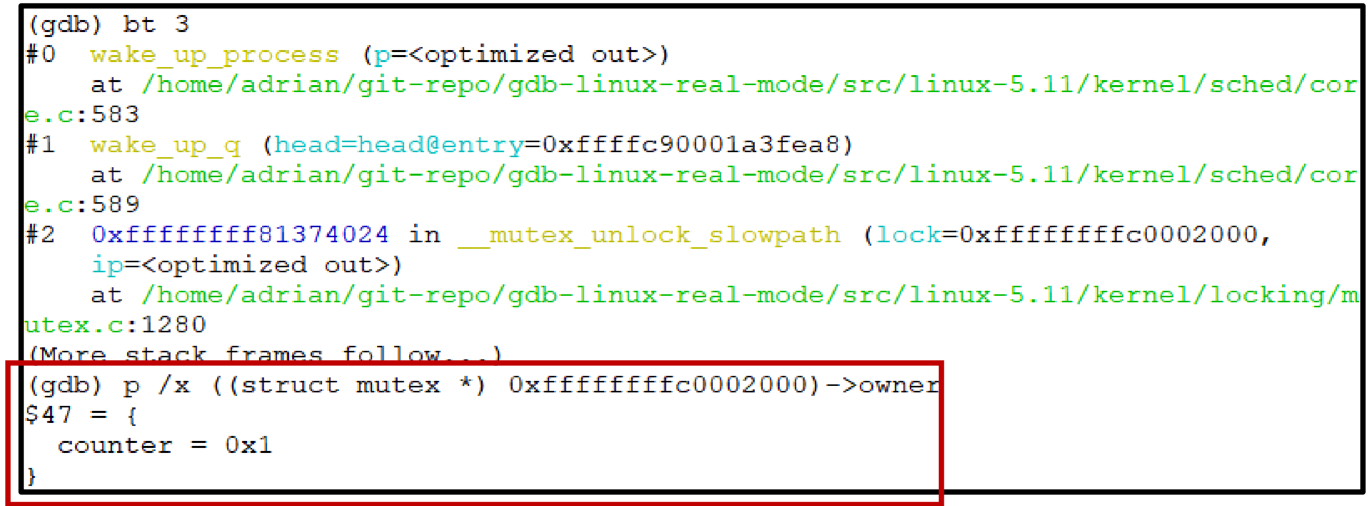
| owner | | | |
|---|---|---|---|
| task virtual addr | P | H | W |
| 63 | 2 | 1 | 0 |

- task_struct pointers aligns to at least L1_CACHE_BYTES
- 3 LSB bits are used for non-empty waiter list
  - ✓ W (MUTEX_FLAG_WAITERS)
    - ■ Non-empty waiter list. Issue a wakeup when unlocking
  - ✓ H (MUTEX_FLAG_HANDOFF)
    - ■ Unlock needs to hand the lock to the top-waiter
    - ■ Use by ww_mutex because ww_mutex's waiter list is not FIFO order.
  - ✓ P (MUTEX_FLAG_PICKUP)
    - ■ Handoff has been done and we're waiting for pickup
    - ■ Use by ww_mutex because ww_mutex's waiter list is not FIFO order.

* ww_mutex (Wound/Wait Mutex): Deadlock-proof mutex

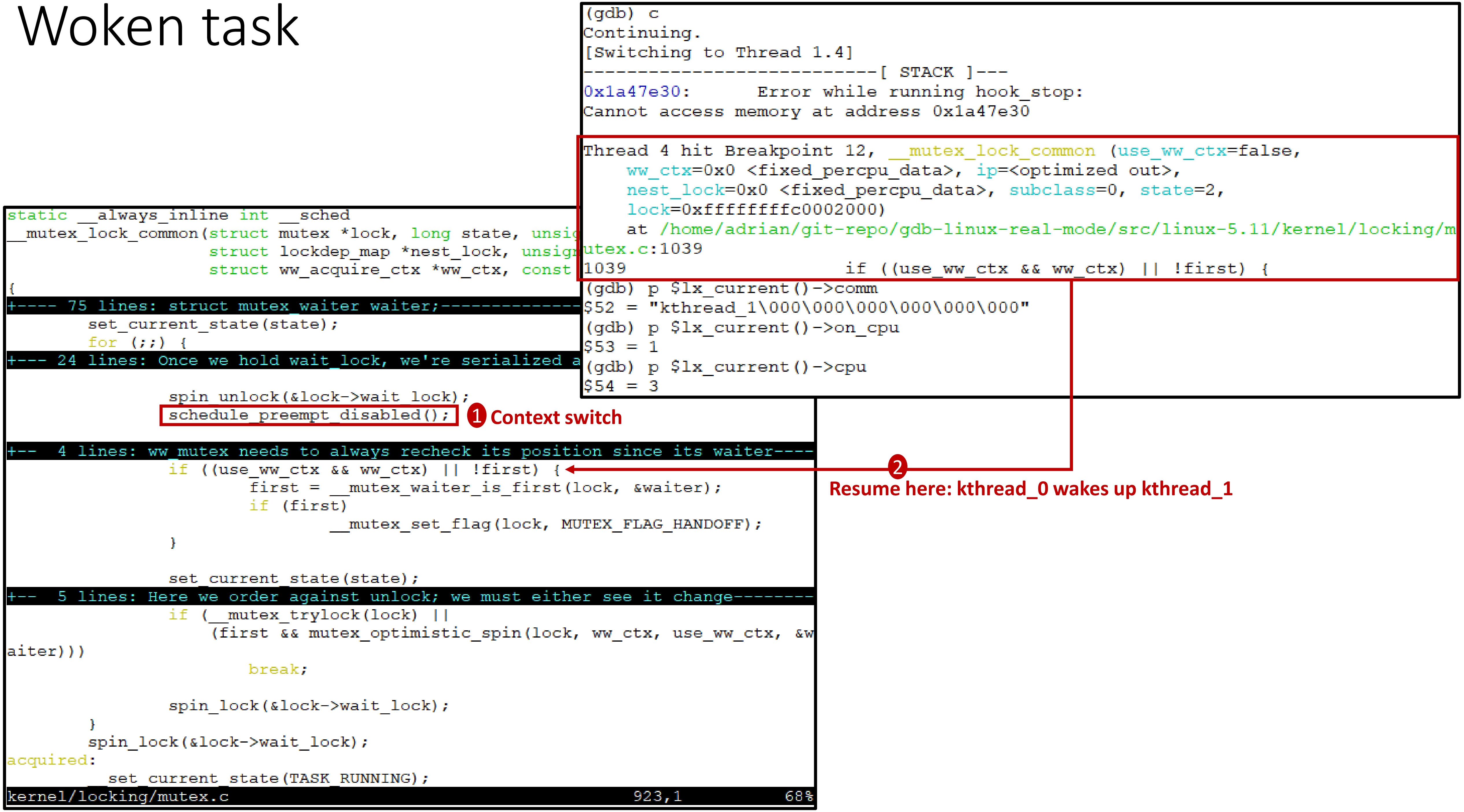
# mutex_unlock(): slow path


mutex_unlock
__mutex_unlock_fast
Have waiters: One of 3-bit LSB of lock-owner is not cleared.
owner
task virtual addr | 0 | 0 | 1
63
2 1 0
[Unlock task = lock->owner]
No waiter: 3-bit LSB of lock->owner are cleared
__mutex_unlock_slowpath
locker->owner = 0
return
atomic_long_cmpxchg_release( &lock->owner, owner, __owner_flags(owner))
owner
0 | 0 | 0 | 1
63
2 1 0
spin_lock(&lock->wait_lock)
Get a waiter from lock->wait_list
spin_unlock(&lock->wait_lock)
__mutex_handoff
Called if MUTEX_FLAG_HANDOFF is set
wake_up_q
wake_up_process
The woken task will update lock->owner



```
(gdb) bt 3
#0  wake_up_process (p=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:583
#1  wake_up_q (head=head@entry=0xffffc90001a3fea8)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/sched/cor
e.c:589
#2  0xffffffff81374024 in __mutex_unlock_slowpath (lock=0xffffffffc0002000,
    ip=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1280
(More stack frames follow...)
(gdb) p /x ((struct mutex *) 0xffffffffc0002000)->owner
$47 = {
  counter = 0x1
}
```

# Woken task



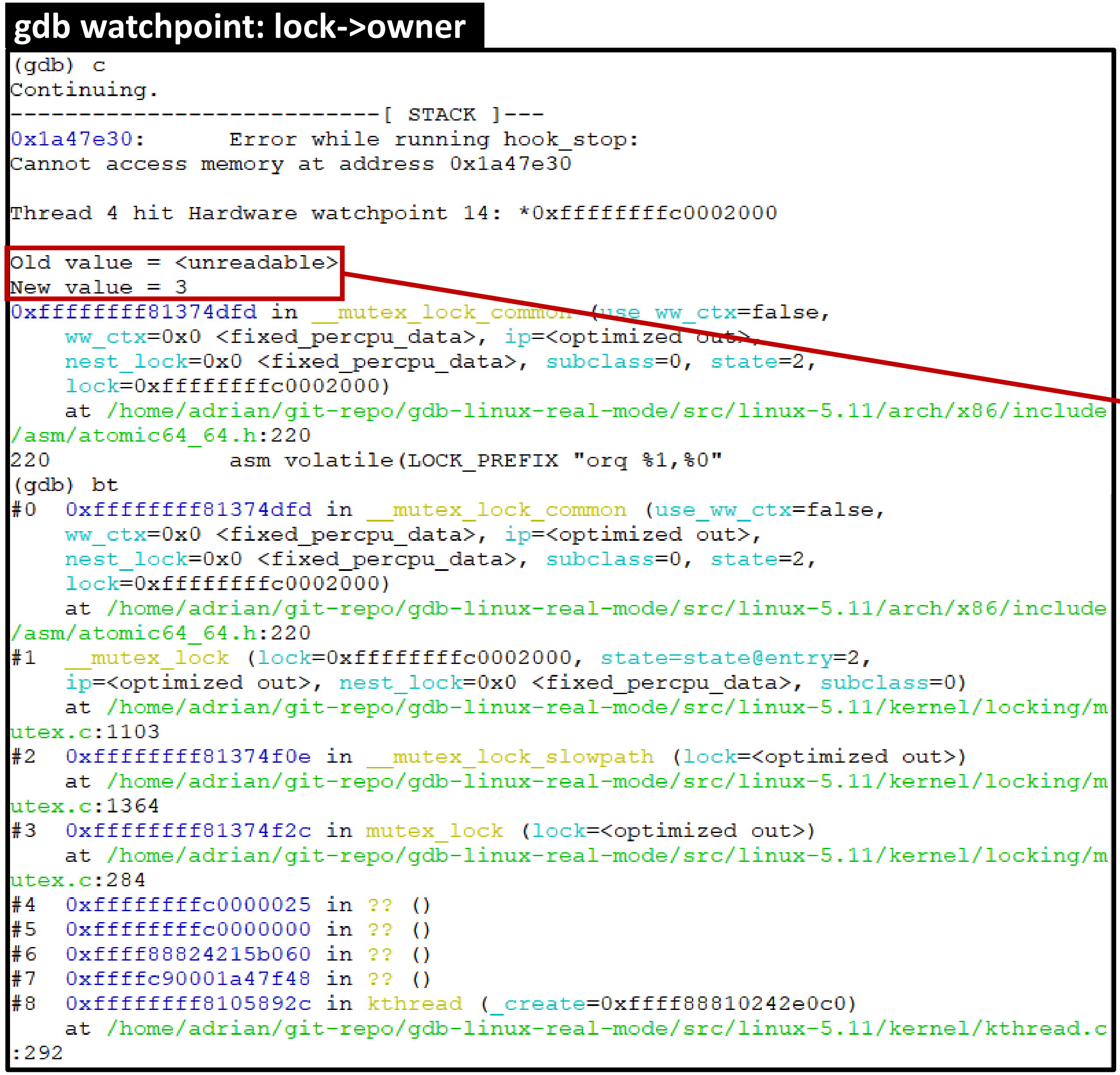
```
static __always_inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsig
                    struct lockdep_map *nest_lock, unsign
                    struct ww_acquire_ctx *ww_ctx, const
{
+---- 75 lines: struct mutex_waiter waiter;---------------
        set_current_state(state);
        for (;;) {
+--- 24 lines: Once we hold wait_lock, we're serialized a

                spin_unlock(&lock->wait_lock);
                schedule_preempt_disabled();

+--  4 lines: ww_mutex needs to always recheck its position since its waiter----
                if ((use_ww_ctx && ww_ctx) || !first) {
                        first = __mutex_waiter_is_first(lock, &waiter);
                        if (first)
                                __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
                }

                set_current_state(state);
+--  5 lines: Here we order against unlock; we must either see it change--------
                if (__mutex_trylock(lock) ||
                    (first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, &w
aiter)))
                        break;

                spin_lock(&lock->wait_lock);
        }
        spin_lock(&lock->wait_lock);
acquired:
        __set_current_state(TASK_RUNNING);
kernel/locking/mutex.c                                          923,1          68%
```

```
(gdb) c
Continuing.
[Switching to Thread 1.4]
-----------------------------[ STACK ]---
0x1a47e30:      Error while running hook_stop:
Cannot access memory at address 0x1a47e30

Thread 4 hit Breakpoint 12, __mutex_lock_common (use_ww_ctx=false,
    ww_ctx=0x0 <fixed_percpu_data>, ip=<optimized out>,
    nest_lock=0x0 <fixed_percpu_data>, subclass=0, state=2,
    lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1039
1039                    if ((use_ww_ctx && ww_ctx) || !first) {
(gdb) p $lx_current()->comm
$52 = "kthread_1\000\000\000\000\000\000"
(gdb) p $lx_current()->on_cpu
$53 = 1
(gdb) p $lx_current()->cpu
$54 = 3
```

1 Context switch

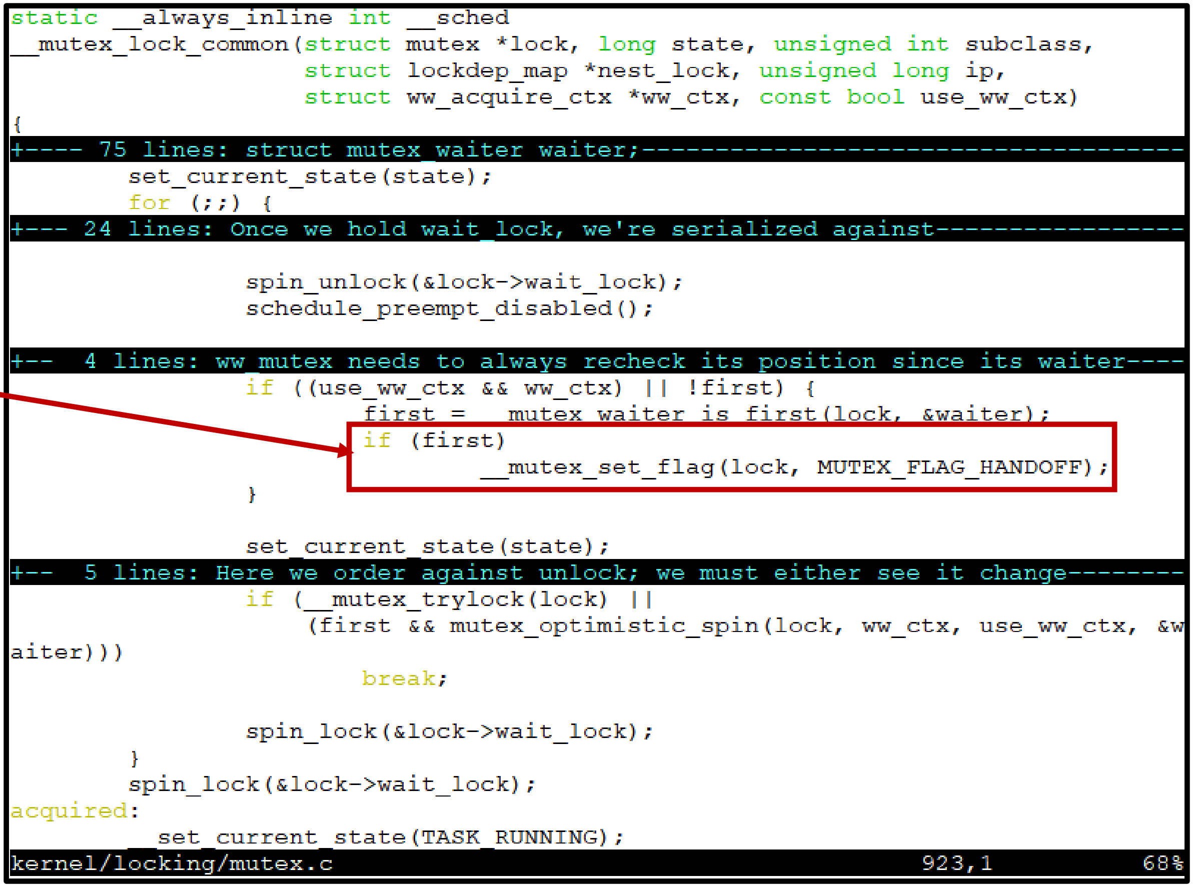
2 Resume here: kthread_0 wakes up kthread_1

# Update lock->owner



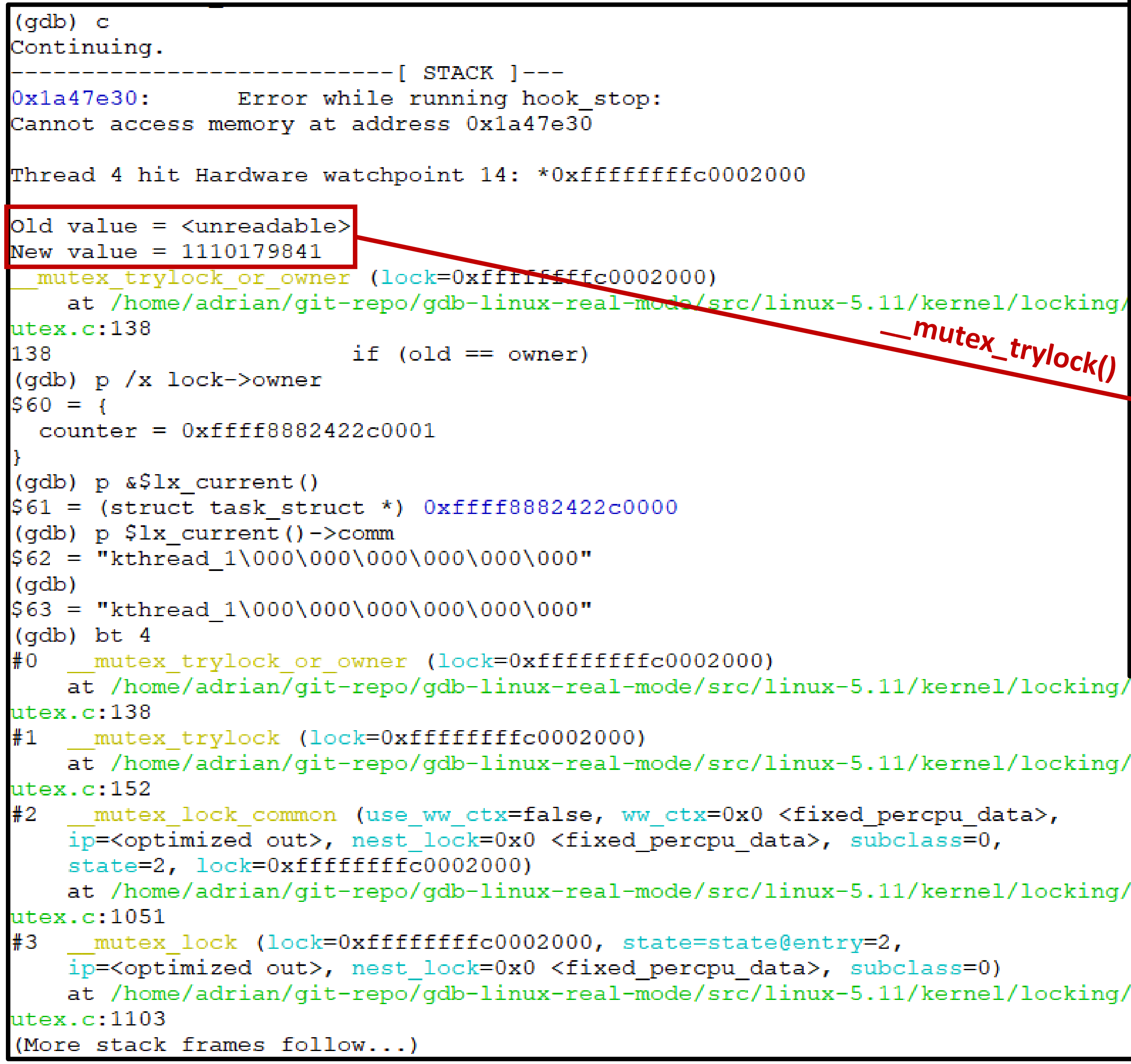
gdb watchpoint: lock->owner

```
(gdb) c
Continuing.
---------------------------[ STACK ]---
0x1a47e30:       Error while running hook_stop:
Cannot access memory at address 0x1a47e30

Thread 4 hit Hardware watchpoint 14: *0xffffffffc0002000

Old value = <unreadable>
New value = 3
0xffffffff81374dfd in __mutex_lock_common (use_ww_ctx=false,
    ww_ctx=0x0 <fixed_percpu_data>, ip=<optimized out>,
    nest_lock=0x0 <fixed_percpu_data>, subclass=0, state=2,
    lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/include
/asm/atomic64_64.h:220
220             asm volatile(LOCK_PREFIX "orq %1,%0"
(gdb) bt
#0  0xffffffff81374dfd in __mutex_lock_common (use_ww_ctx=false,
    ww_ctx=0x0 <fixed_percpu_data>, ip=<optimized out>,
    nest_lock=0x0 <fixed_percpu_data>, subclass=0, state=2,
    lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/arch/x86/include
/asm/atomic64_64.h:220
#1  __mutex_lock (lock=0xffffffffc0002000, state=state@entry=2,
    ip=<optimized out>, nest_lock=0x0 <fixed_percpu_data>, subclass=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1103
#2  0xffffffff81374f0e in __mutex_lock_slowpath (lock=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1364
#3  0xffffffff81374f2c in mutex_lock (lock=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:284
#4  0xffffffffc0000025 in ?? ()
#5  0xffffffffc0000000 in ?? ()
#6  0xffff88824215b060 in ?? ()
#7  0xffffc90001a47f48 in ?? ()
#8  0xffffffff8105892c in kthread (_create=0xffff88810242e0c0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/kthread.c
:292
```



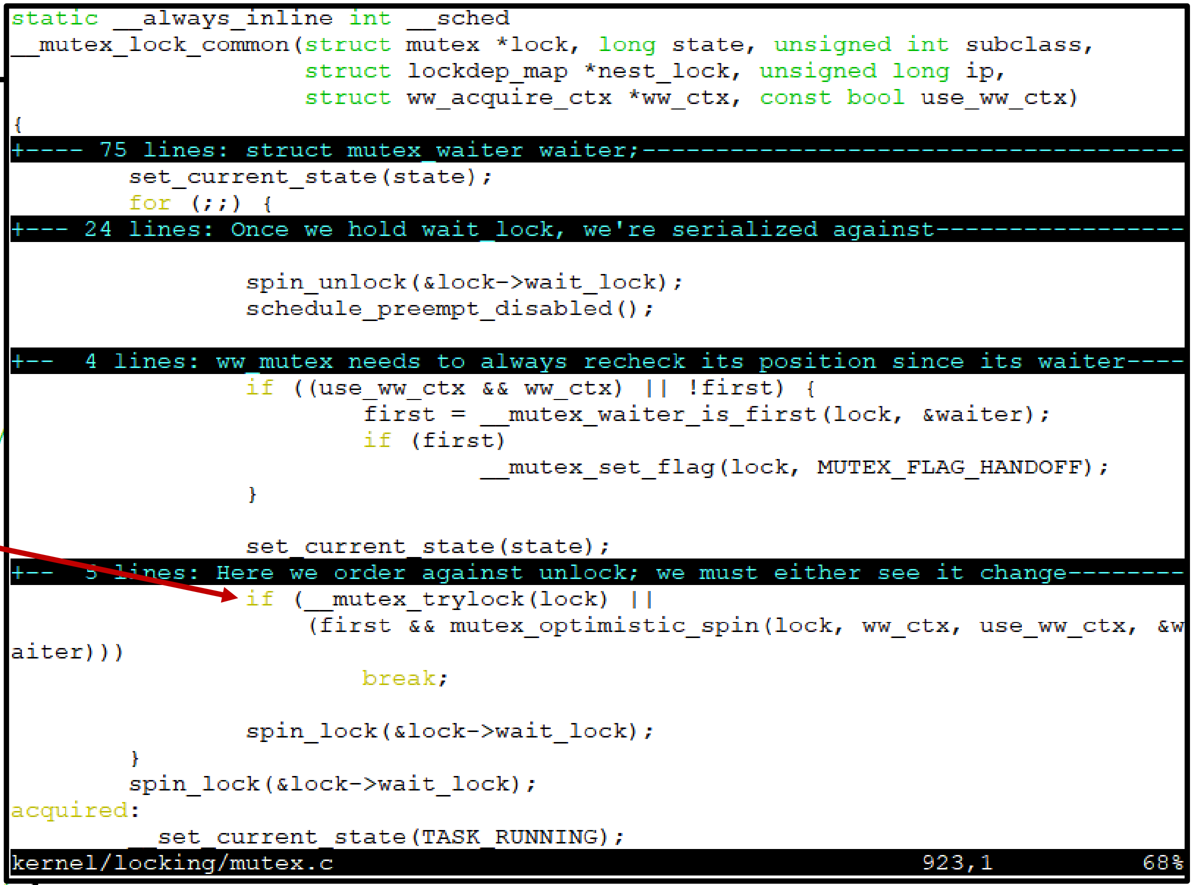
```
static __always_inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
                    struct lockdep_map *nest_lock, unsigned long ip,
                    struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
{
+---- 75 lines: struct mutex_waiter waiter;------------------------------------
        set_current_state(state);
        for (;;) {
+--- 24 lines: Once we hold wait_lock, we're serialized against-----------------

                spin_unlock(&lock->wait_lock);
                schedule_preempt_disabled();

+--  4 lines: ww_mutex needs to always recheck its position since its waiter----
                if ((use_ww_ctx && ww_ctx) || !first) {
                        first = __mutex_waiter_is_first(lock, &waiter);
                        if (first)
                                __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
                }

                set_current_state(state);
+--  5 lines: Here we order against unlock; we must either see it change--------
                if (__mutex_trylock(lock) ||
                    (first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, &w
aiter)))
                        break;

                spin_lock(&lock->wait_lock);
        }
        spin_lock(&lock->wait_lock);
acquired:
        __set_current_state(TASK_RUNNING);
kernel/locking/mutex.c                                    923,1          68%
```

# Update lock->owner



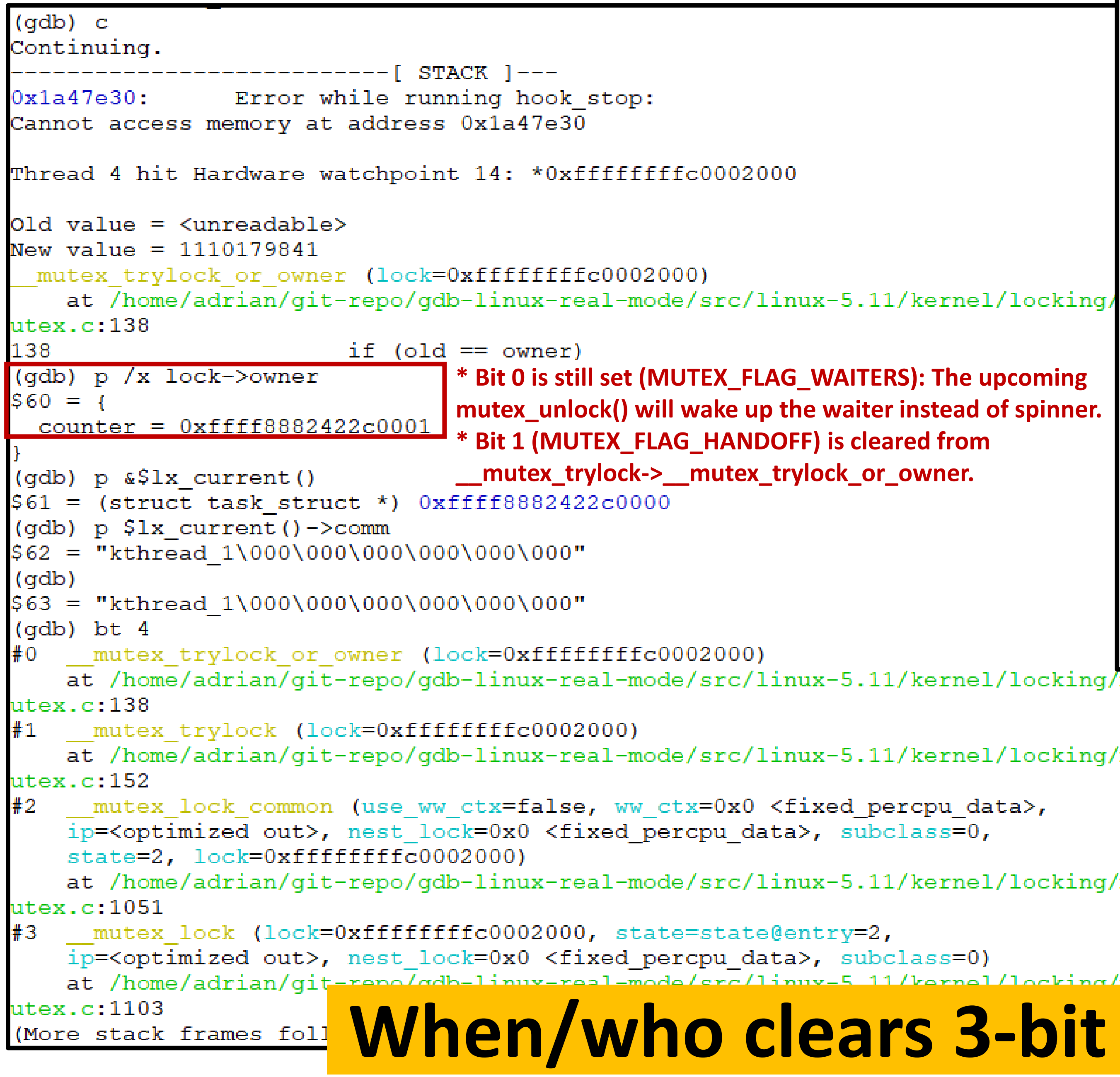
```
(gdb) c
Continuing.
---------------------------[ STACK ]---
0x1a47e30:       Error while running hook_stop:
Cannot access memory at address 0x1a47e30

Thread 4 hit Hardware watchpoint 14: *0xffffffffc0002000

Old value = <unreadable>
New value = 1110179841
__mutex_trylock_or_owner (lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/
utex.c:138
138                     if (old == owner)
(gdb) p /x lock->owner
$60 = {
  counter = 0xffff8882422c0001
}
(gdb) p &$lx_current()
$61 = (struct task_struct *) 0xffff8882422c0000
(gdb) p $lx_current()->comm
$62 = "kthread_1\000\000\000\000\000\000"
(gdb)
$63 = "kthread_1\000\000\000\000\000\000"
(gdb) bt 4
#0  __mutex_trylock_or_owner (lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:138
#1  __mutex_trylock (lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:152
#2  __mutex_lock_common (use_ww_ctx=false, ww_ctx=0x0 <fixed_percpu_data>,
    ip=<optimized out>, nest_lock=0x0 <fixed_percpu_data>, subclass=0,
    state=2, lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1051
#3  __mutex_lock (lock=0xffffffffc0002000, state=state@entry=2,
    ip=<optimized out>, nest_lock=0x0 <fixed_percpu_data>, subclass=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1103
(More stack frames follow...)
```

__mutex_trylock()



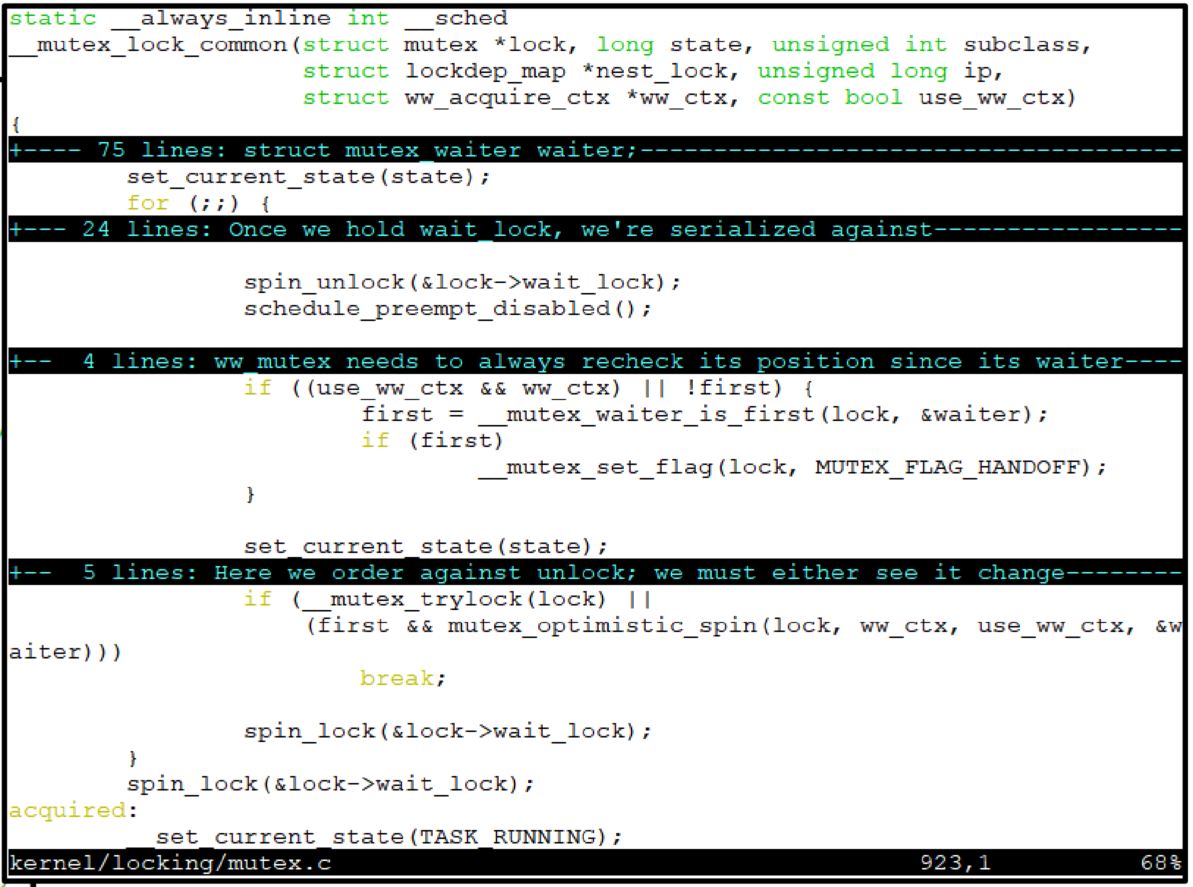
```
static __always_inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
                    struct lockdep_map *nest_lock, unsigned long ip,
                    struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
{
+---- 75 lines: struct mutex_waiter waiter;--------------------------------------
        set_current_state(state);
        for (;;) {
+--- 24 lines: Once we hold wait_lock, we're serialized against-----------------

                spin_unlock(&lock->wait_lock);
                schedule_preempt_disabled();

+--  4 lines: ww_mutex needs to always recheck its position since its waiter----
                if ((use_ww_ctx && ww_ctx) || !first) {
                        first = __mutex_waiter_is_first(lock, &waiter);
                        if (first)
                                __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
                }

                set_current_state(state);
+--  5 lines: Here we order against unlock; we must either see it change--------
                if (__mutex_trylock(lock) ||
                    (first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, &w
aiter)))
                        break;

                spin_lock(&lock->wait_lock);
        }
        spin_lock(&lock->wait_lock);
acquired:
        __set_current_state(TASK_RUNNING);
kernel/locking/mutex.c                                             923,1          68%
```

# Update lock->owner



```
(gdb) c
Continuing.
---------------------------[ STACK ]---
0x1a47e30:       Error while running hook_stop:
Cannot access memory at address 0x1a47e30

Thread 4 hit Hardware watchpoint 14: *0xffffffffc0002000

Old value = <unreadable>
New value = 1110179841
__mutex_trylock_or_owner (lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/
utex.c:138
138                     if (old == owner)
(gdb) p /x lock->owner
$60 = {
  counter = 0xffff8882422c0001
}
(gdb) p &$lx_current()
$61 = (struct task_struct *) 0xffff8882422c0000
(gdb) p $lx_current()->comm
$62 = "kthread_1\000\000\000\000\000\000"
(gdb)
$63 = "kthread_1\000\000\000\000\000\000"
(gdb) bt 4
#0  __mutex_trylock_or_owner (lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:138
#1  __mutex_trylock (lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:152
#2  __mutex_lock_common (use_ww_ctx=false, ww_ctx=0x0 <fixed_percpu_data>,
    ip=<optimized out>, nest_lock=0x0 <fixed_percpu_data>, subclass=0,
    state=2, lock=0xffffffffc0002000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1051
#3  __mutex_lock (lock=0xffffffffc0002000, state=state@entry=2,
    ip=<optimized out>, nest_lock=0x0 <fixed_percpu_data>, subclass=0)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1103
(More stack frames foll
```

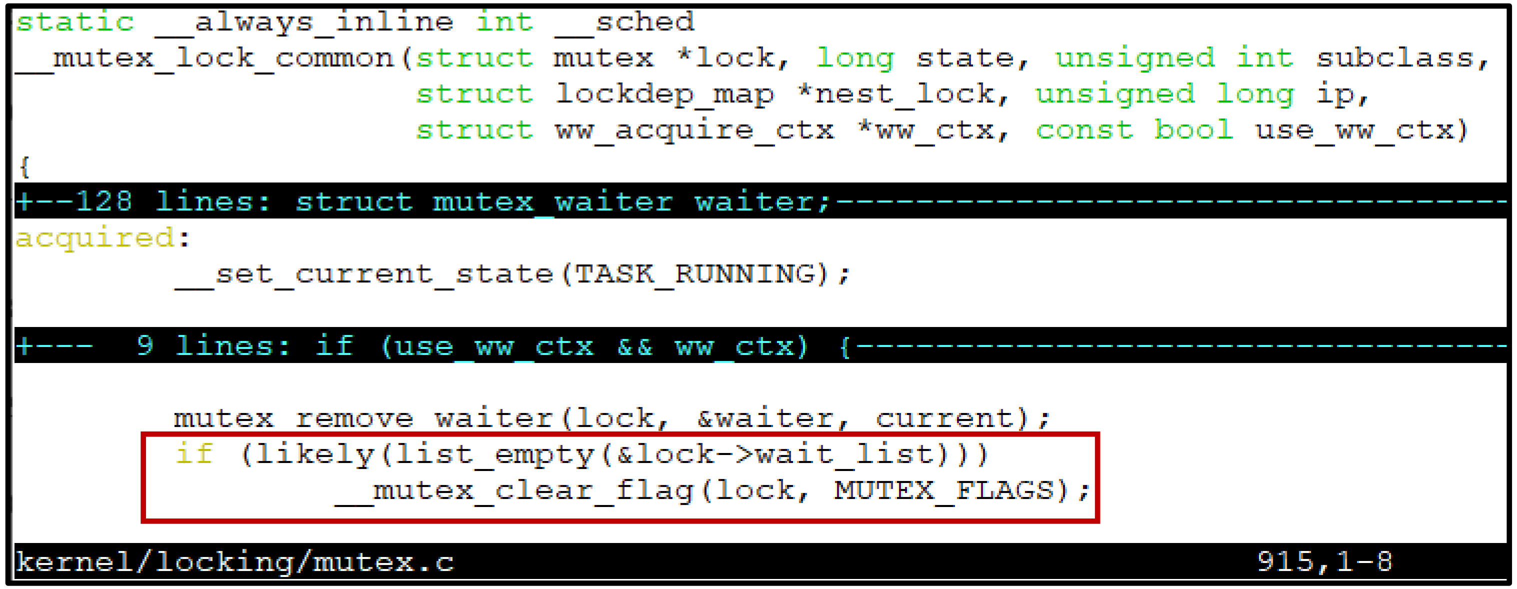
* Bit 0 is still set (MUTEX_FLAG_WAITERS): The upcoming mutex_unlock() will wake up the waiter instead of spinner.
* Bit 1 (MUTEX_FLAG_HANDOFF) is cleared from __mutex_trylock->__mutex_trylock_or_owner.



```
static __always_inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
                    struct lockdep_map *nest_lock, unsigned long ip,
                    struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
{
+---- 75 lines: struct mutex_waiter waiter;------------------------------------
        set_current_state(state);
        for (;;) {
+--- 24 lines: Once we hold wait_lock, we're serialized against----------------

                spin_unlock(&lock->wait_lock);
                schedule_preempt_disabled();

+--  4 lines: ww_mutex needs to always recheck its position since its waiter----
                if ((use_ww_ctx && ww_ctx) || !first) {
                        first = __mutex_waiter_is_first(lock, &waiter);
                        if (first)
                                __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
                }

                set_current_state(state);
+--  5 lines: Here we order against unlock; we must either see it change-------
                if (__mutex_trylock(lock) ||
                    (first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, &w
aiter)))
                        break;

                spin_lock(&lock->wait_lock);
        }
        spin_lock(&lock->wait_lock);
acquired:
        __set_current_state(TASK_RUNNING);
kernel/locking/mutex.c                                        923,1          68%
```

**When/who clears 3-bit LSB of lock->owner?**

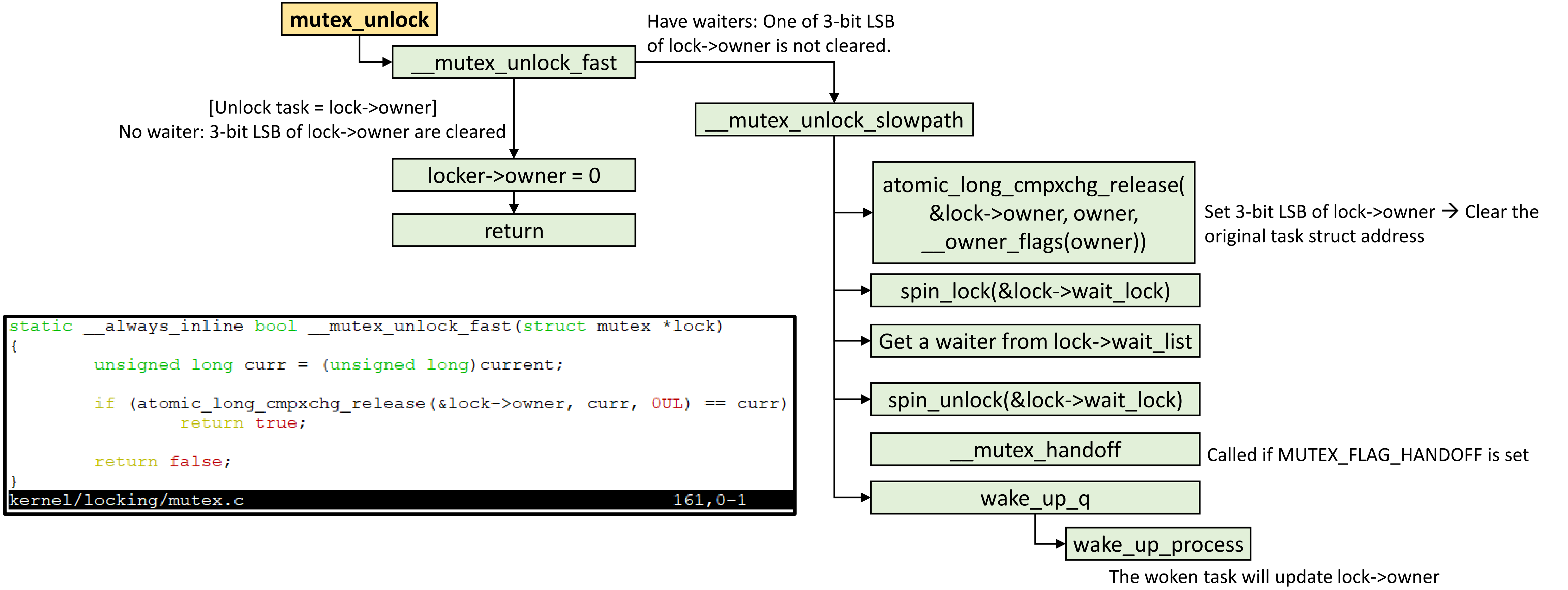
# Woken task: When/who to clear 3-bit LSB of lock-owner?



```
static __always_inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
                    struct lockdep_map *nest_lock, unsigned long ip,
                    struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
{
+--128 lines: struct mutex_waiter waiter;-----------------------------------
acquired:
        __set_current_state(TASK_RUNNING);

+---  9 lines: if (use_ww_ctx && ww_ctx) {----------------------------------

        mutex_remove_waiter(lock, &waiter, current);
        if (likely(list_empty(&lock->wait_list)))
                __mutex_clear_flag(lock, MUTEX_FLAGS);

kernel/locking/mutex.c                                              915,1-8
```

**Clear 3-bit LSB of lock->owner if no waiters**

# mutex_unlock(): Mutex ownership



mutex_unlock

__mutex_unlock_fast

Have waiters: One of 3-bit LSB of lock->owner is not cleared.

[Unlock task = lock->owner]
No waiter: 3-bit LSB of lock->owner are cleared

locker->owner = 0

return

__mutex_unlock_slowpath

atomic_long_cmpxchg_release( &lock->owner, owner, __owner_flags(owner))

Set 3-bit LSB of lock->owner → Clear the original task struct address

spin_lock(&lock->wait_lock)

Get a waiter from lock->wait_list

spin_unlock(&lock->wait_lock)

__mutex_handoff

Called if MUTEX_FLAG_HANDOFF is set

wake_up_q

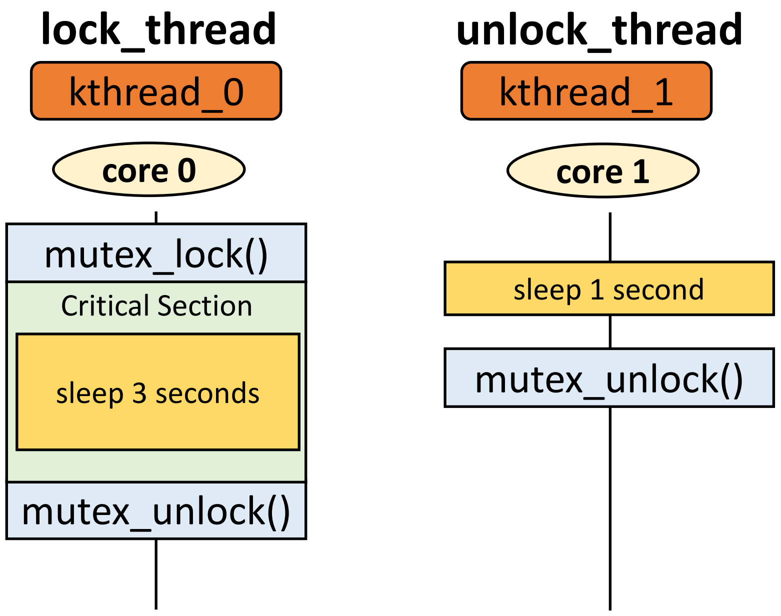
wake_up_process

The woken task will update lock->owner

```
static __always_inline bool __mutex_unlock_fast(struct mutex *lock)
{
        unsigned long curr = (unsigned long)current;

        if (atomic_long_cmpxchg_release(&lock->owner, curr, 0UL) == curr)
                return true;

        return false;
}
kernel/locking/mutex.c                                          161,0-1
```

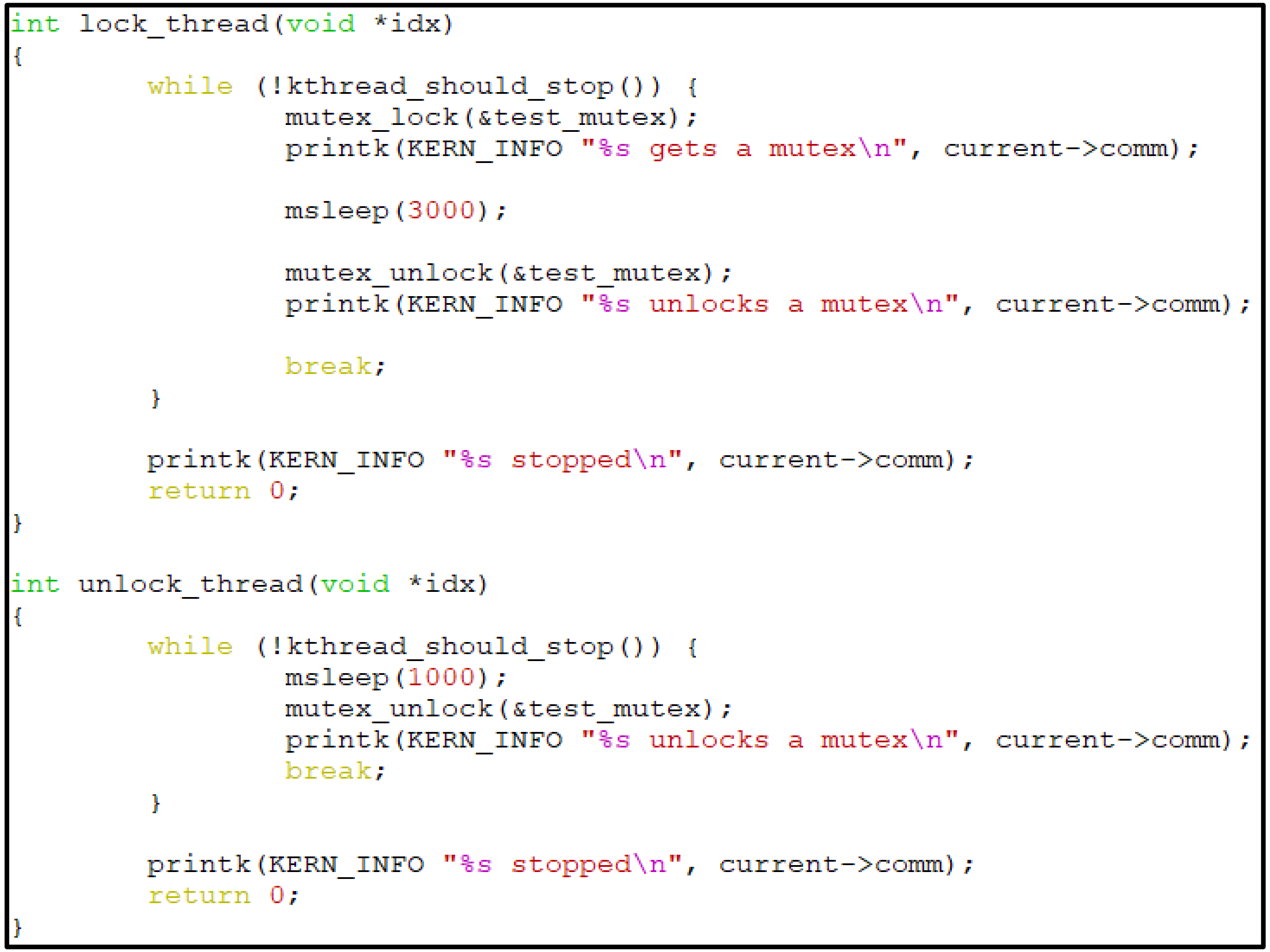
- [Fastpath] Check ownership of a mutex
- [Slowpath] Does not check ownership of a mutex

# mutex_unlock(): [lab] Behavior observation when lock owner != unlocker’s task (Mutex ownership)



**Note**

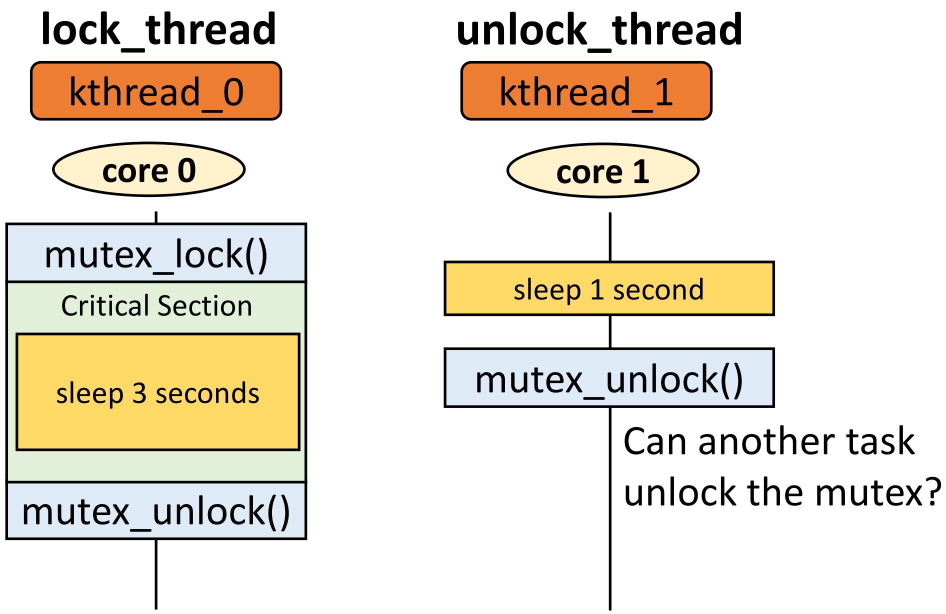
This scenario is created on purpose for demonstration. It won’t happen in real case.



```
int lock_thread(void *idx)
{
        while (!kthread_should_stop()) {
                mutex_lock(&test_mutex);
                printk(KERN_INFO "%s gets a mutex\n", current->comm);

                msleep(3000);

                mutex_unlock(&test_mutex);
                printk(KERN_INFO "%s unlocks a mutex\n", current->comm);

                break;
        }

        printk(KERN_INFO "%s stopped\n", current->comm);
        return 0;
}

int unlock_thread(void *idx)
{
        while (!kthread_should_stop()) {
                msleep(1000);
                mutex_unlock(&test_mutex);
                printk(KERN_INFO "%s unlocks a mutex\n", current->comm);
                break;
        }

        printk(KERN_INFO "%s stopped\n", current->comm);
        return 0;
}
```

Source code: test-modules/mutex-unlock-by-another-task/mutex.c

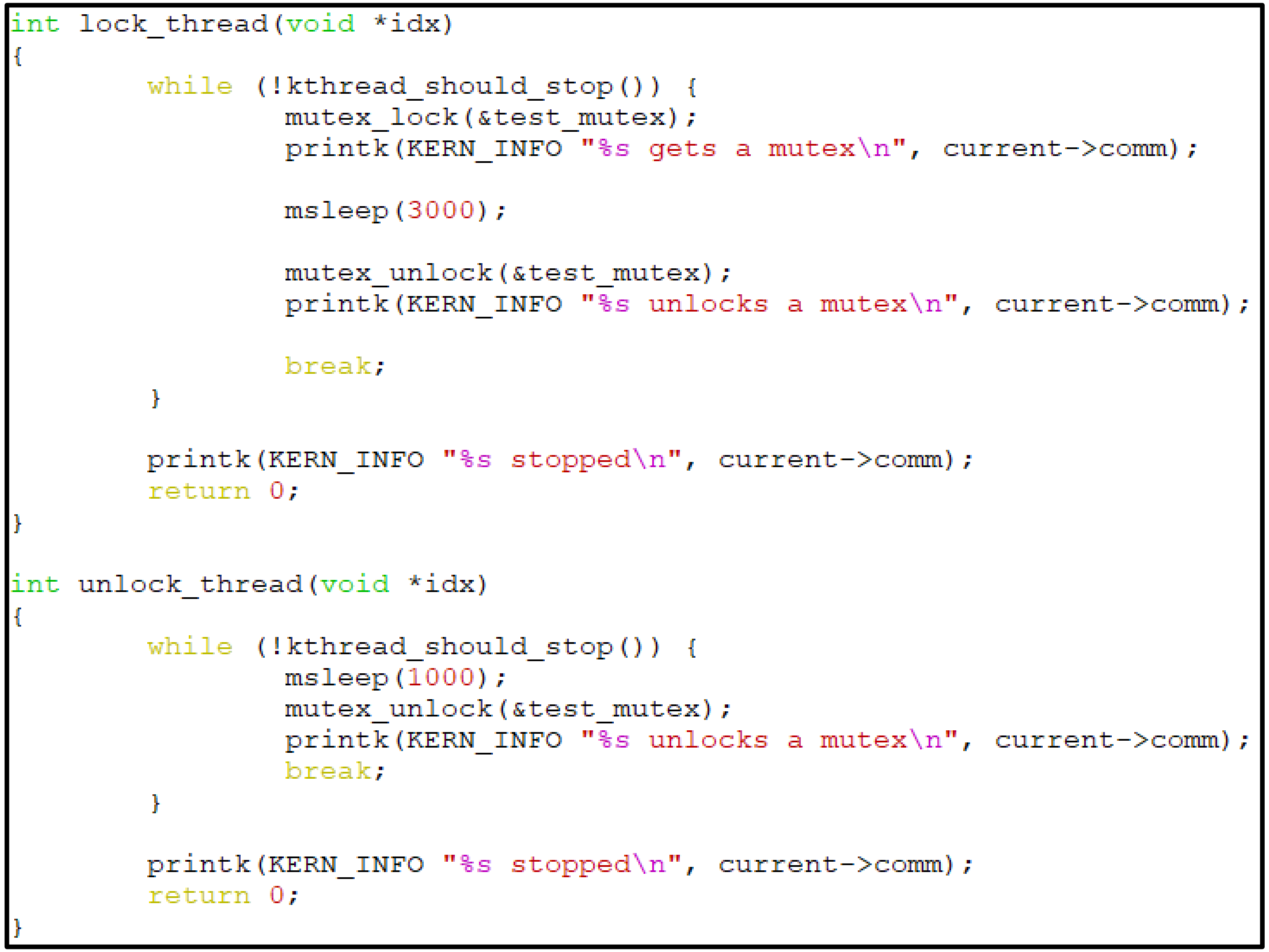
# mutex_unlock(): [lab] Behavior observation when lock owner != unlocker's task (Mutex ownership)



Can another task unlock the mutex?

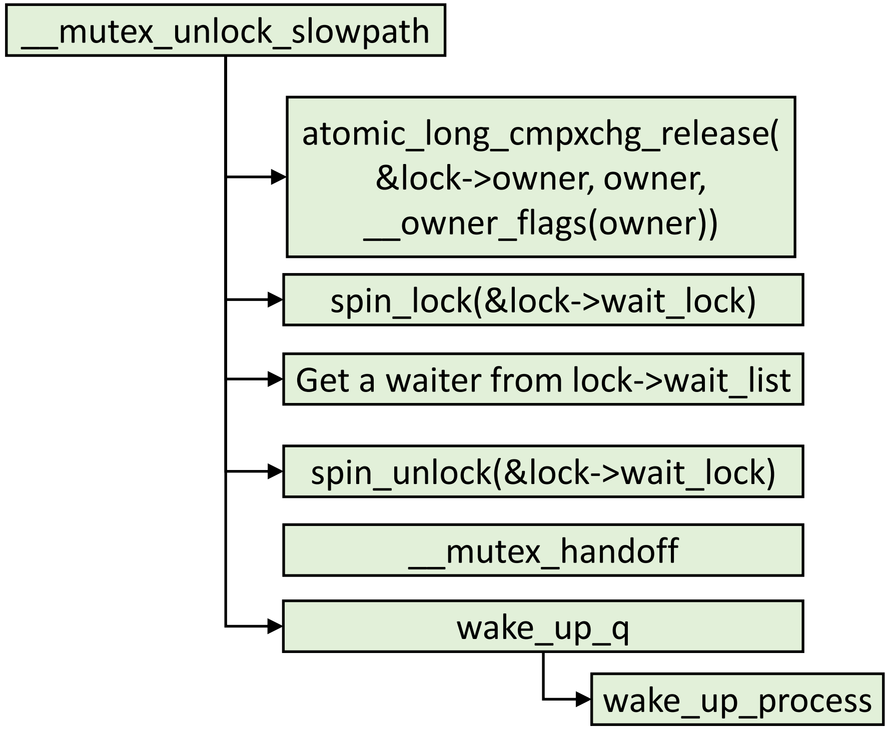
**Note**

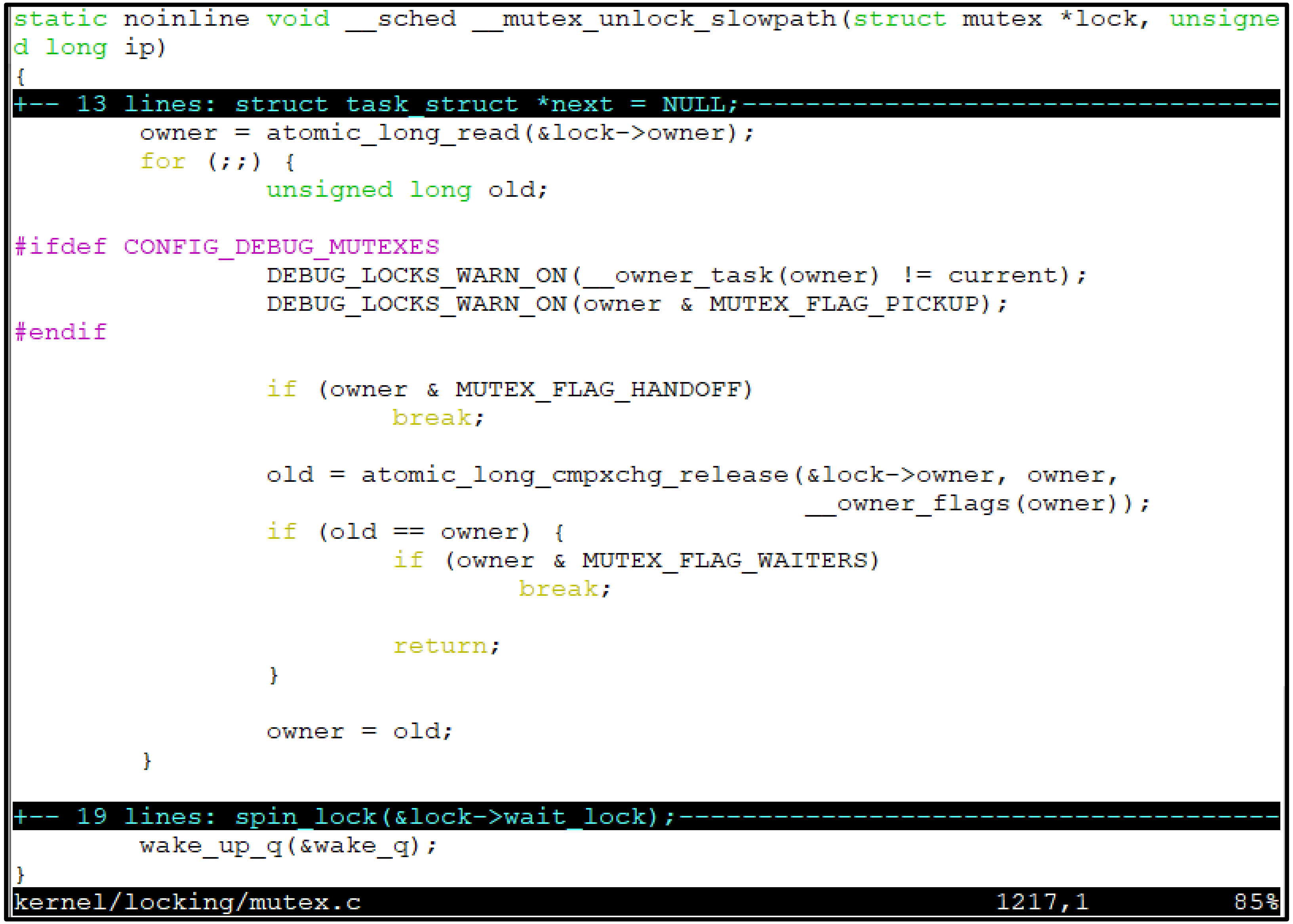
This scenario is created on purpose for demonstration. It won't happen in real case.



```
int lock_thread(void *idx)
{
        while (!kthread_should_stop()) {
                mutex_lock(&test_mutex);
                printk(KERN_INFO "%s gets a mutex\n", current->comm);

                msleep(3000);

                mutex_unlock(&test_mutex);
                printk(KERN_INFO "%s unlocks a mutex\n", current->comm);

                break;
        }

        printk(KERN_INFO "%s stopped\n", current->comm);
        return 0;
}

int unlock_thread(void *idx)
{
        while (!kthread_should_stop()) {
                msleep(1000);
                mutex_unlock(&test_mutex);
                printk(KERN_INFO "%s unlocks a mutex\n", current->comm);
                break;
        }

        printk(KERN_INFO "%s stopped\n", current->comm);
        return 0;
}
```

# mutex_unlock(): slow path does not check unlocker's ownership

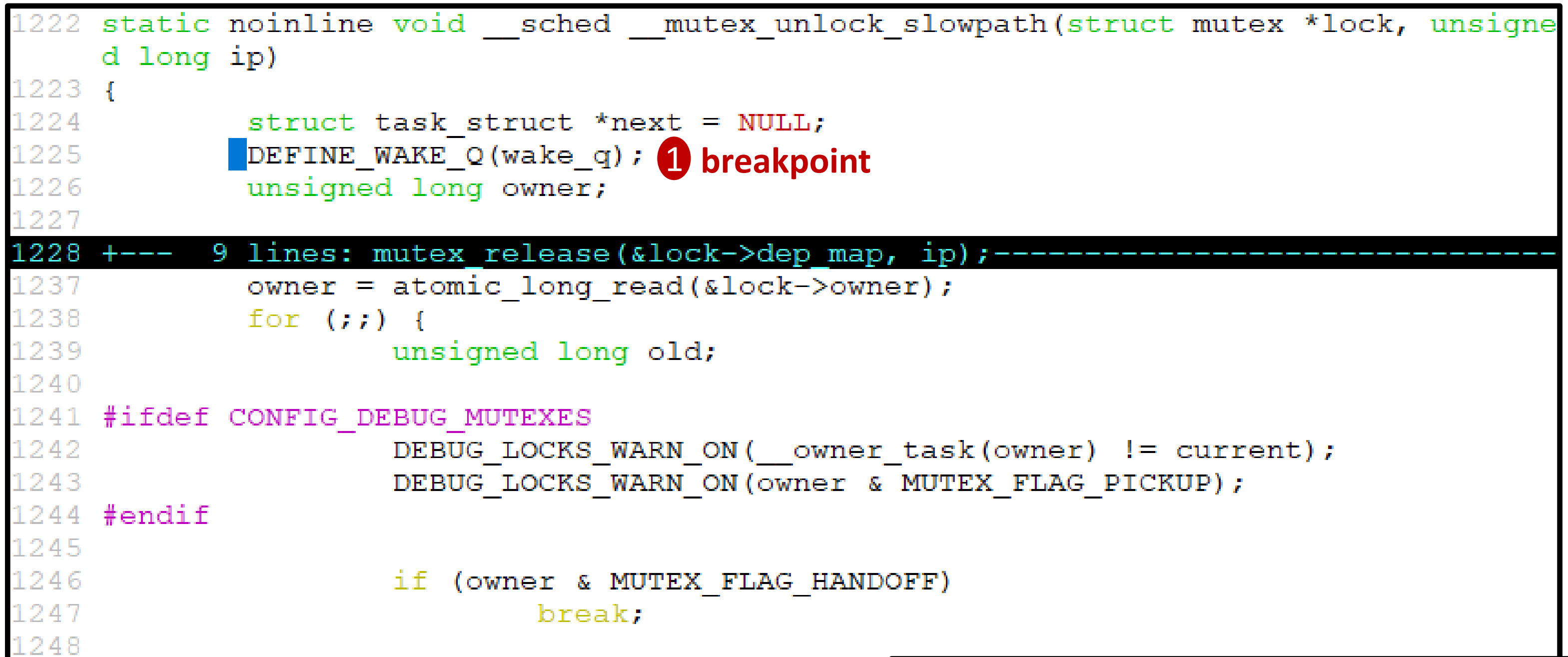

__mutex_unlock_slowpath
atomic_long_cmpxchg_release( &lock->owner, owner, __owner_flags(owner))
spin_lock(&lock->wait_lock)
Get a waiter from lock->wait_list
spin_unlock(&lock->wait_lock)
__mutex_handoff
wake_up_q
wake_up_process



```
static noinline void __sched __mutex_unlock_slowpath(struct mutex *lock, unsigne
d long ip)
{
+-- 13 lines: struct task_struct *next = NULL;----------------------------------
        owner = atomic_long_read(&lock->owner);
        for (;;) {
                unsigned long old;

#ifdef CONFIG_DEBUG_MUTEXES
                DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current);
                DEBUG_LOCKS_WARN_ON(owner & MUTEX_FLAG_PICKUP);
#endif

                if (owner & MUTEX_FLAG_HANDOFF)
                        break;

                old = atomic_long_cmpxchg_release(&lock->owner, owner,
                                                  __owner_flags(owner));
                if (old == owner) {
                        if (owner & MUTEX_FLAG_WAITERS)
                                break;

                        return;
                }

                owner = old;
        }

+-- 19 lines: spin_lock(&lock->wait_lock);--------------------------------------
        wake_up_q(&wake_q);
}
kernel/locking/mutex.c                                              1217,1   85%
```

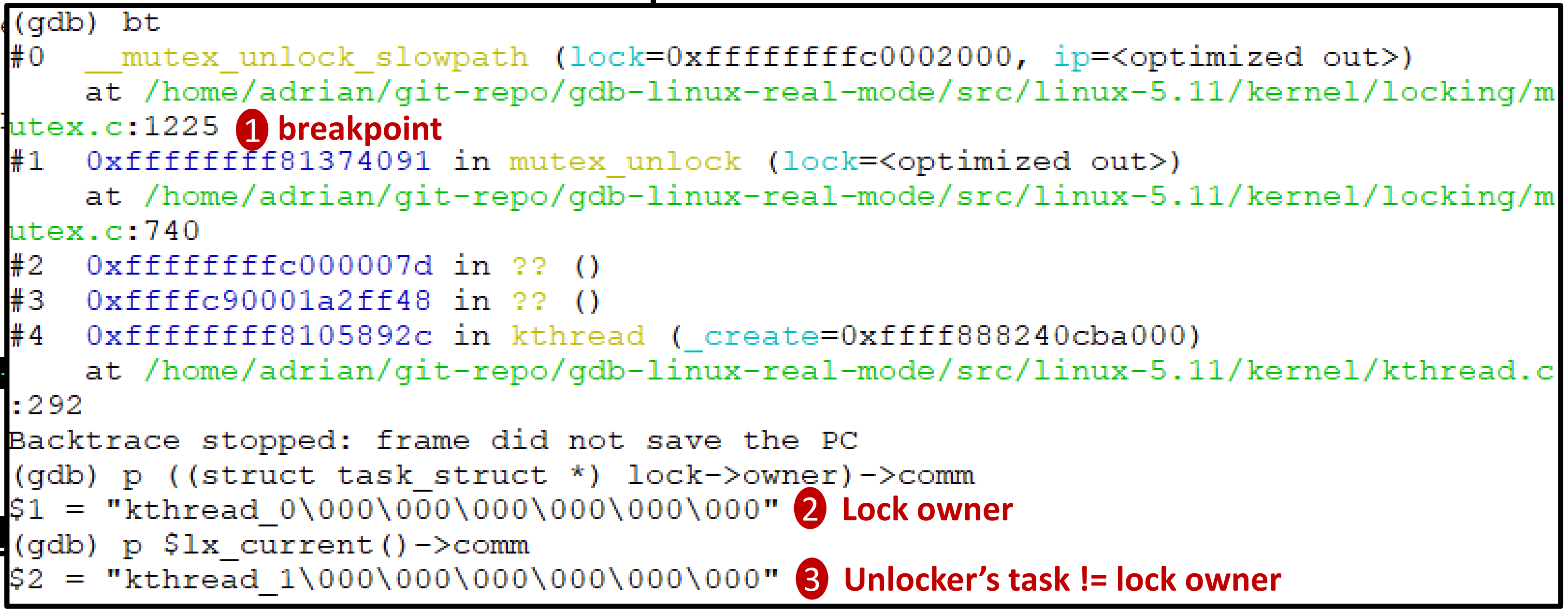
**[DEBUG_MUTEXES] Print a warning message if unlocker's task != lock owner's task**

# mutex_unlock(): [lab] Behavior when lock owner != unlocker’s task



```
static noinline void __sched __mutex_unlock_slowpath(struct mutex *lock, unsigned long ip)
{
        struct task_struct *next = NULL;
        DEFINE_WAKE_Q(wake_q);
        unsigned long owner;

+---  9 lines: mutex_release(&lock->dep_map, ip);-----------------------------------
        owner = atomic_long_read(&lock->owner);
        for (;;) {
                unsigned long old;

#ifdef CONFIG_DEBUG_MUTEXES
                DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current);
                DEBUG_LOCKS_WARN_ON(owner & MUTEX_FLAG_PICKUP);
#endif

                if (owner & MUTEX_FLAG_HANDOFF)
                        break;

                old = atomic_long_cmpxchg_re

                if (old == owner) {
                        if (owner & MUTEX_F
                                break;

                        return;
                }

                owner = old;
        }
+-- 19 lines: spin_lock(&lock->wait_lock);-
        
        wake_up_q(&wake_q);
}

kernel/locking/mutex.c
```

1 breakpoint

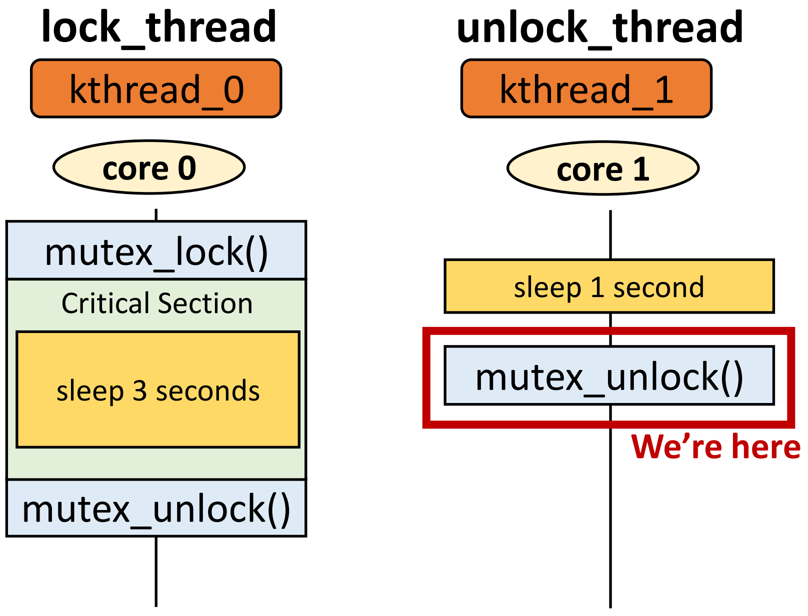


```
(gdb) bt
#0  __mutex_unlock_slowpath (lock=0xffffffffc0002000, ip=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/mutex.c:1225
#1  0xffffffff81374091 in mutex_unlock (lock=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/mutex.c:740
#2  0xffffffffc000007d in ?? ()
#3  0xffffc90001a2ff48 in ?? ()
#4  0xffffffff8105892c in kthread (_create=0xffff888240cba000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/kthread.c:292
Backtrace stopped: frame did not save the PC
(gdb) p ((struct task_struct *) lock->owner)->comm
$1 = "kthread_0\000\000\000\000\000\000"
(gdb) p $lx_current()->comm
$2 = "kthread_1\000\000\000\000\000\000"
```

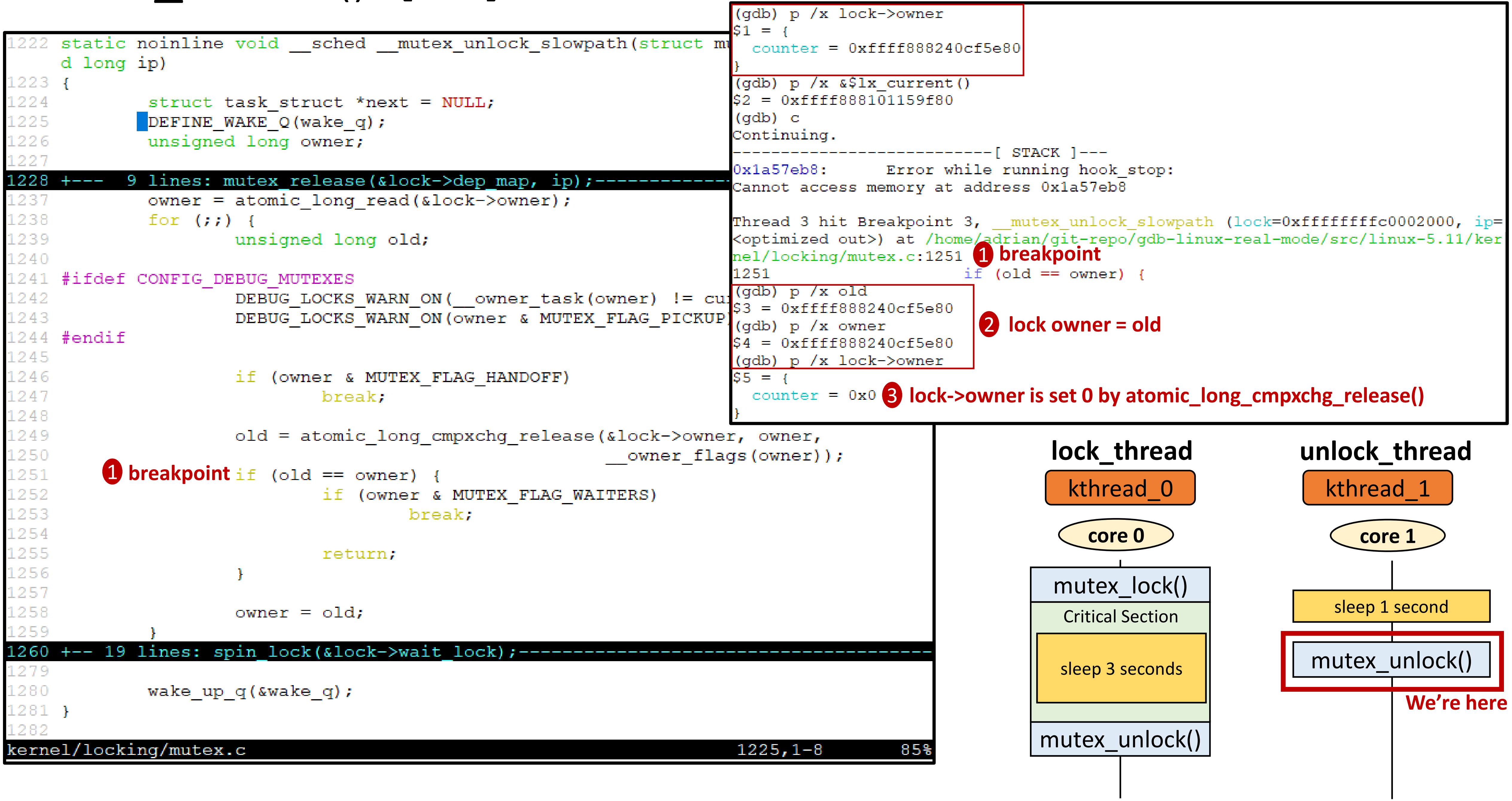
1 breakpoint

2 Lock owner

3 Unlocker’s task != lock owner



# mutex_unlock(): [lab] Behavior if lock owner != unlocker’s task



```
1222 static noinline void __sched __mutex_unlock_slowpath(struct mu
     d long ip)
1223 {
1224         struct task_struct *next = NULL;
1225         DEFINE_WAKE_Q(wake_q);
1226         unsigned long owner;
1227
1228 +---  9 lines: mutex_release(&lock->dep_map, ip);-------------
1237         owner = atomic_long_read(&lock->owner);
1238         for (;;) {
1239                 unsigned long old;
1240
1241 #ifdef CONFIG_DEBUG_MUTEXES
1242                 DEBUG_LOCKS_WARN_ON(__owner_task(owner) != cu
1243                 DEBUG_LOCKS_WARN_ON(owner & MUTEX_FLAG_PICKUP)
1244 #endif
1245
1246                 if (owner & MUTEX_FLAG_HANDOFF)
1247                         break;
1248
1249                 old = atomic_long_cmpxchg_release(&lock->owner, owner,
1250                                                   __owner_flags(owner));
1251                 if (old == owner) {
1252                         if (owner & MUTEX_FLAG_WAITERS)
1253                                 break;
1254
1255                         return;
1256                 }
1257
1258                 owner = old;
1259         }
1260 +-- 19 lines: spin_lock(&lock->wait_lock);-----------------------------------------
1279
1280         wake_up_q(&wake_q);
1281 }
1282
kernel/locking/mutex.c                                       1225,1-8      85%
```

❶ breakpoint

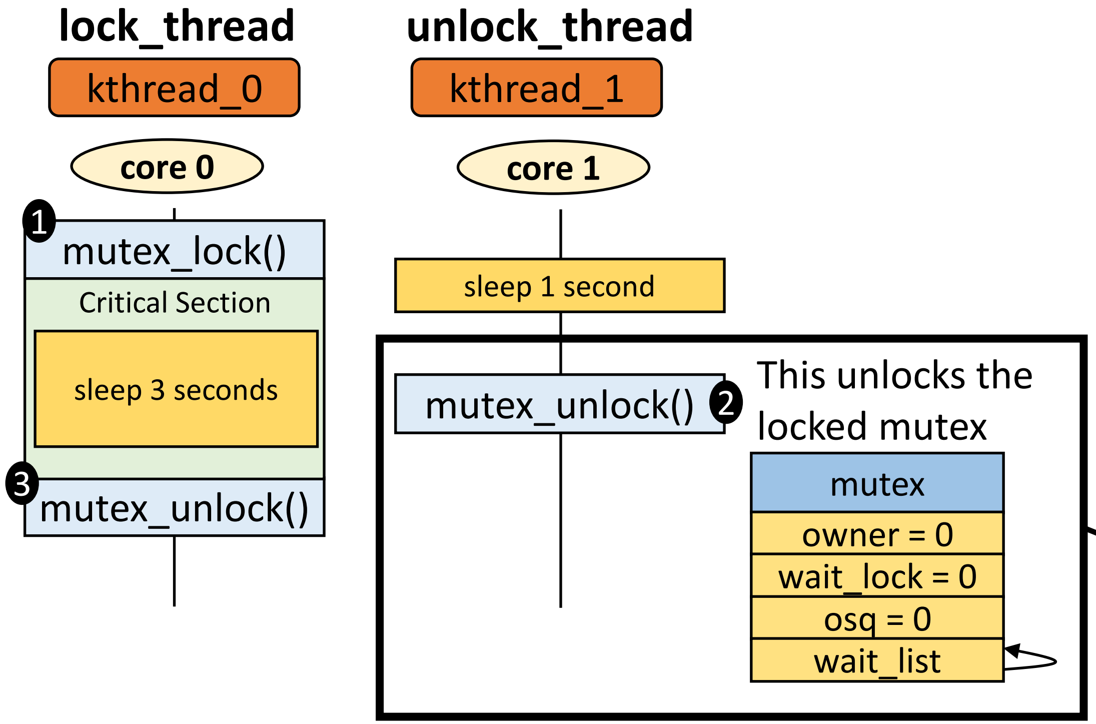
```
(gdb) p /x lock->owner
$1 = {
  counter = 0xffff888240cf5e80
}
(gdb) p /x &$lx_current()
$2 = 0xffff888101159f80
(gdb) c
Continuing.
---------------------------[ STACK ]---
0x1a57eb8:      Error while running hook_stop:
Cannot access memory at address 0x1a57eb8

Thread 3 hit Breakpoint 3, __mutex_unlock_slowpath (lock=0xffffffffc0002000, ip=
<optimized out>) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/ker
nel/locking/mutex.c:1251
1251                    if (old == owner) {
(gdb) p /x old
$3 = 0xffff888240cf5e80
(gdb) p /x owner
$4 = 0xffff888240cf5e80
(gdb) p /x lock->owner
$5 = {
  counter = 0x0
}
```

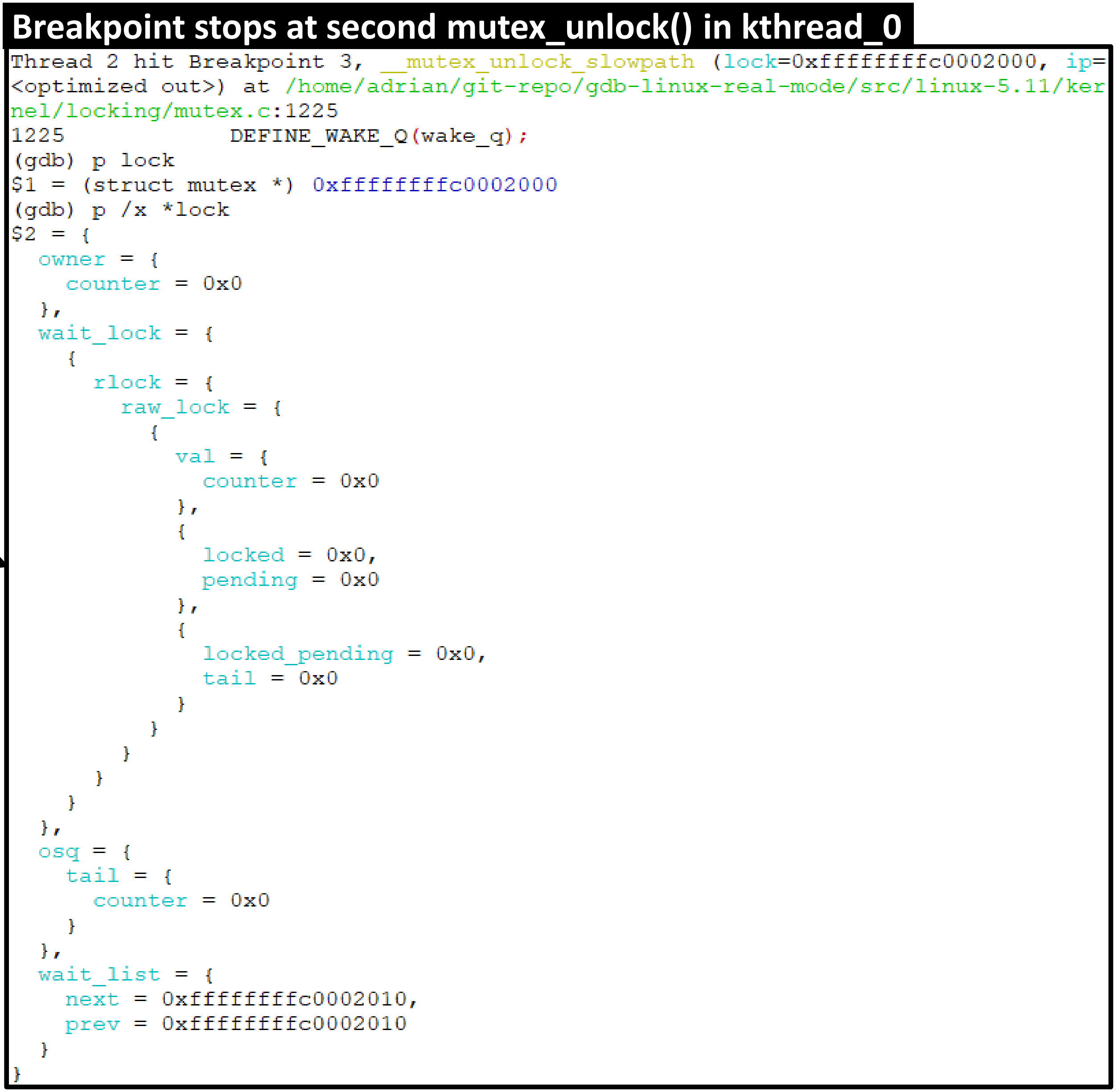
❶ breakpoint

❷ lock owner = old

❸ lock->owner is set 0 by atomic_long_cmpxchg_release()

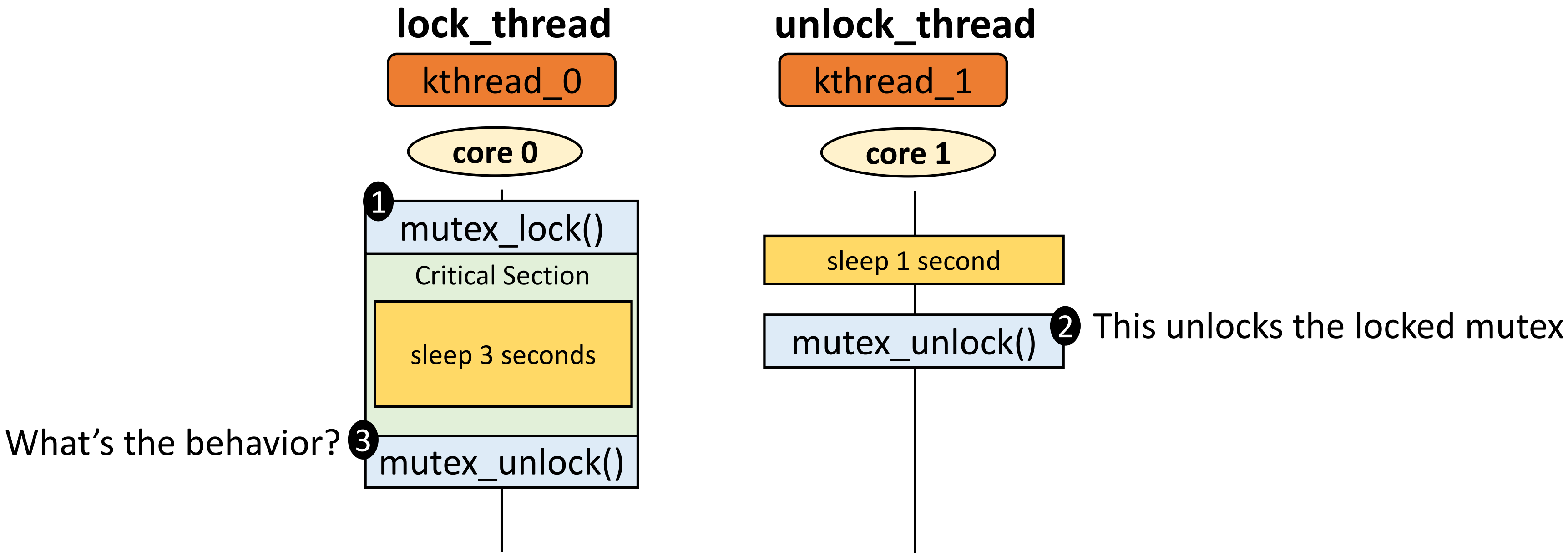
# mutex_unlock(): [lab] Behavior observation when lock owner != unlocker’s task (Mutex ownership)




**Breakpoint stops at second mutex_unlock() in kthread_0**

```
Thread 2 hit Breakpoint 3, __mutex_unlock_slowpath (lock=0xffffffffc0002000, ip=
<optimized out>) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/ker
nel/locking/mutex.c:1225
1225            DEFINE_WAKE_Q(wake_q);
(gdb) p lock
$1 = (struct mutex *) 0xffffffffc0002000
(gdb) p /x *lock
$2 = {
  owner = {
    counter = 0x0
  },
  wait_lock = {
    {
      rlock = {
        raw_lock = {
          {
            val = {
              counter = 0x0
            },
            {
              locked = 0x0,
              pending = 0x0
            },
            {
              locked_pending = 0x0,
              tail = 0x0
            }
          }
        }
      }
    }
  },
  osq = {
    tail = {
      counter = 0x0
    }
  },
  wait_list = {
    next = 0xffffffffc0002010,
    prev = 0xffffffffc0002010
  }
}
```

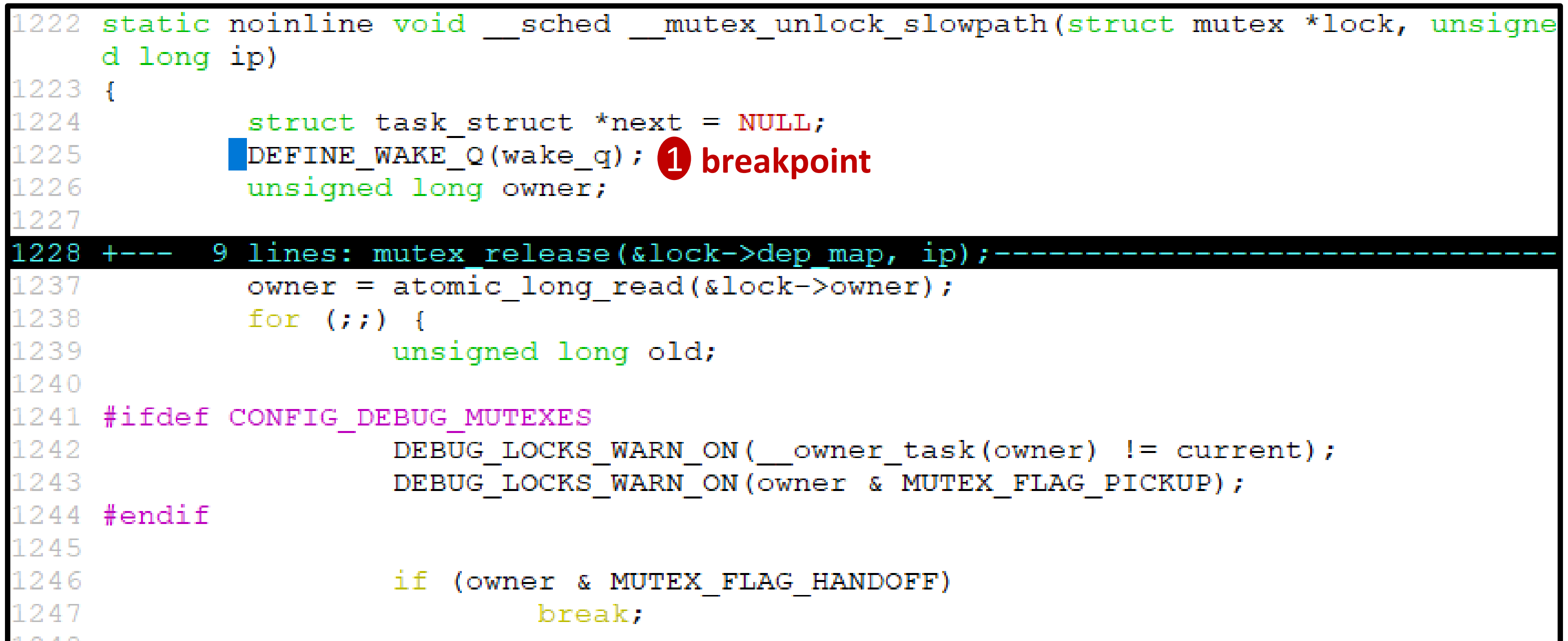
# mutex_unlock(): [lab] Behavior observation when lock owner != unlocker's task (Mutex ownership)



This unlocks the locked mutex

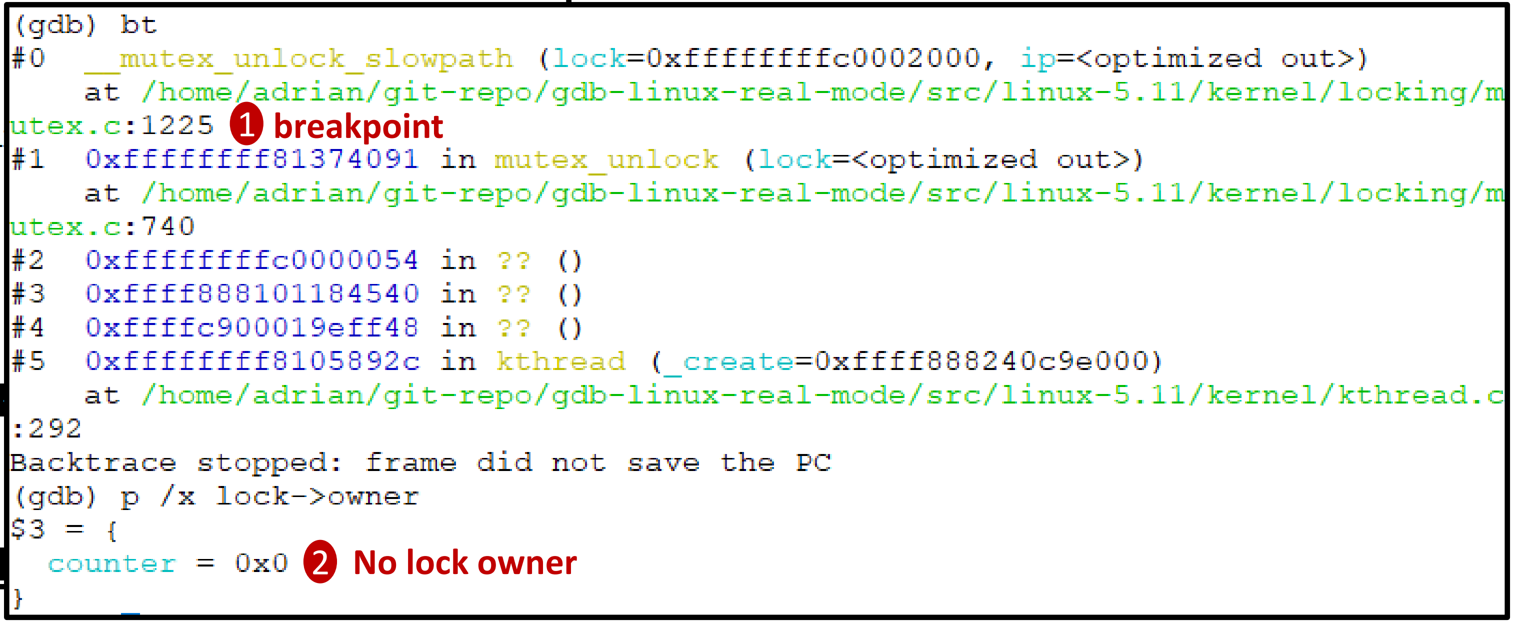
What's the behavior?

# mutex_unlock(): [lab] Behavior when lock owner != unlocker's task



```
1222 static noinline void __sched __mutex_unlock_slowpath(struct mutex *lock, unsigne
     d long ip)
1223 {
1224         struct task_struct *next = NULL;
1225         DEFINE_WAKE_Q(wake_q);
1226         unsigned long owner;
1227
1228 +---  9 lines: mutex_release(&lock->dep_map, ip);--------------------------------
1237         owner = atomic_long_read(&lock->owner);
1238         for (;;) {
1239                 unsigned long old;
1240
1241 #ifdef CONFIG_DEBUG_MUTEXES
1242                 DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current);
1243                 DEBUG_LOCKS_WARN_ON(owner & MUTEX_FLAG_PICKUP);
1244 #endif
1245
1246                 if (owner & MUTEX_FLAG_HANDOFF)
1247                         break;
1248
1249                 old = atomic_long_cmpxchg_release
1250
1251                 if (old == owner) {
1252                         if (owner & MUTEX_FLAG_WA
1253                                 break;
1254
1255                         return;
1256                 }
1257
1258                 owner = old;
1259         }
1260 +-- 19 lines: spin_lock(&lock->wait_lock);-------
1279
1280         wake_up_q(&wake_q);
1281 }
1282
kernel/locking/mutex.c
```

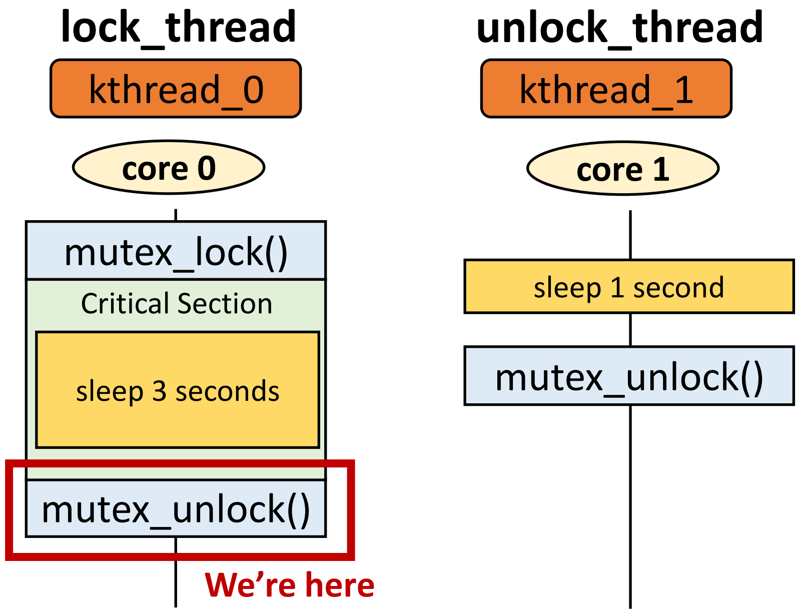
❶ breakpoint



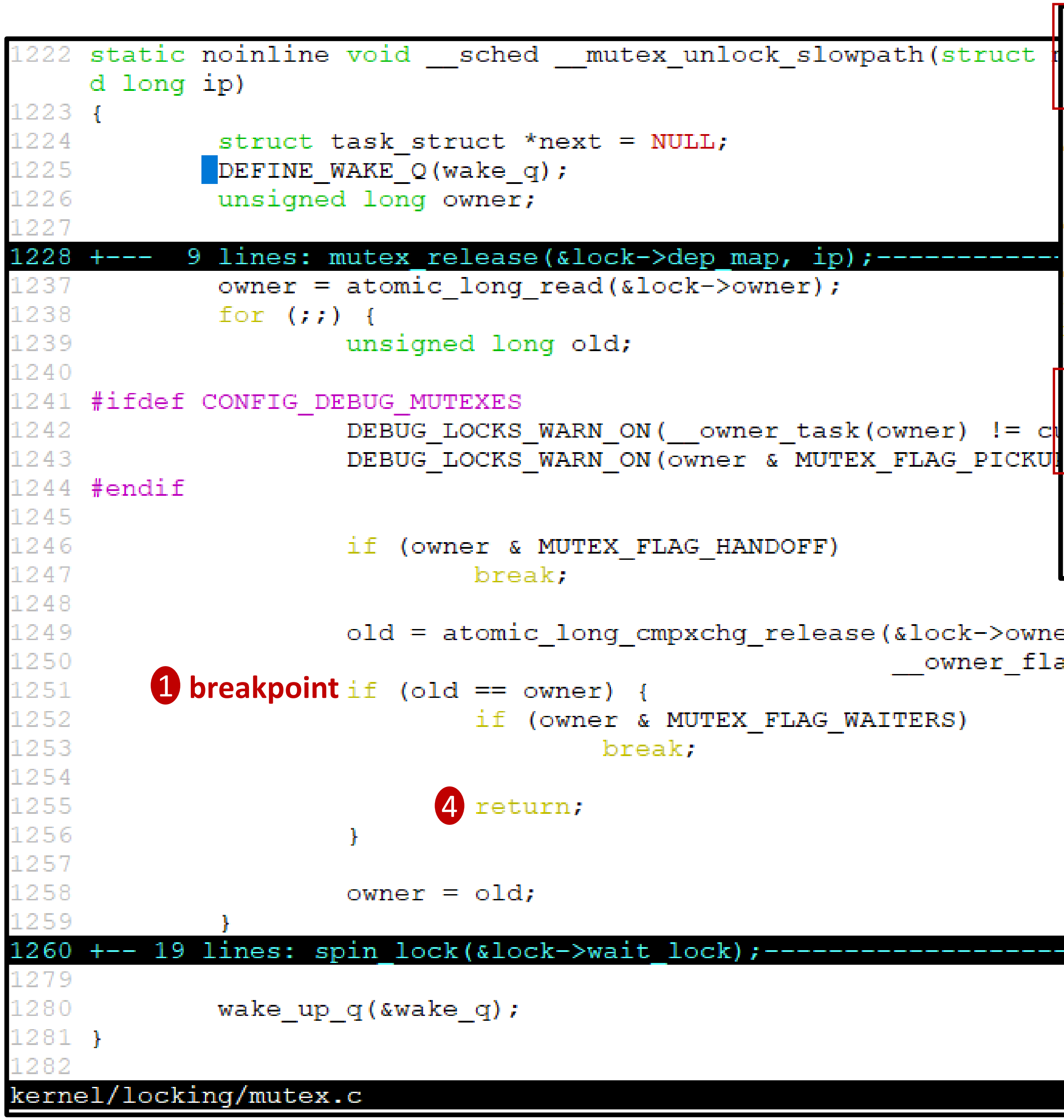
```
(gdb) bt
#0  __mutex_unlock_slowpath (lock=0xffffffffc0002000, ip=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:1225
#1  0xffffffff81374091 in mutex_unlock (lock=<optimized out>)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/locking/m
utex.c:740
#2  0xffffffffc0000054 in ?? ()
#3  0xffff888101184540 in ?? ()
#4  0xffffc900019eff48 in ?? ()
#5  0xffffffff8105892c in kthread (_create=0xffff888240c9e000)
    at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/kernel/kthread.c
:292
Backtrace stopped: frame did not save the PC
(gdb) p /x lock->owner
$3 = {
  counter = 0x0
}
```

❶ breakpoint

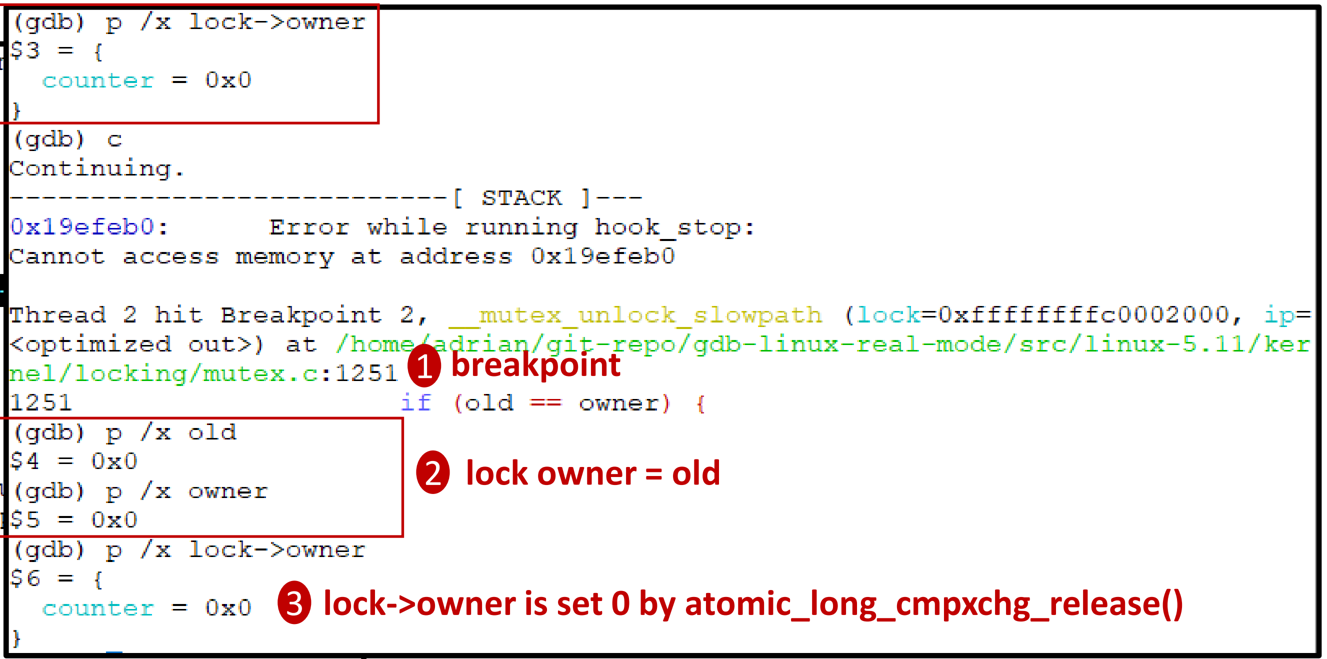
❷ No lock owner



# mutex_unlock(): [lab] Behavior if lock owner != unlocker's task



```
static noinline void __sched __mutex_unlock_slowpath(struct 
d long ip)
{
        struct task_struct *next = NULL;
        DEFINE_WAKE_Q(wake_q);
        unsigned long owner;

+---  9 lines: mutex_release(&lock->dep_map, ip);-----------
        owner = atomic_long_read(&lock->owner);
        for (;;) {
                unsigned long old;

#ifdef CONFIG_DEBUG_MUTEXES
                DEBUG_LOCKS_WARN_ON(__owner_task(owner) != c
                DEBUG_LOCKS_WARN_ON(owner & MUTEX_FLAG_PICKU
#endif

                if (owner & MUTEX_FLAG_HANDOFF)
                        break;

                old = atomic_long_cmpxchg_release(&lock->owner, owner,
                                                  __owner_flags(owner));
      1 breakpoint if (old == owner) {
                        if (owner & MUTEX_FLAG_WAITERS)
                                break;

                     4 return;
                }

                owner = old;
        }
+-- 19 lines: spin_lock(&lock->wait_lock);-----------------------------------
        wake_up_q(&wake_q);
}

kernel/locking/mutex.c                                    1225,1-8       85%
```

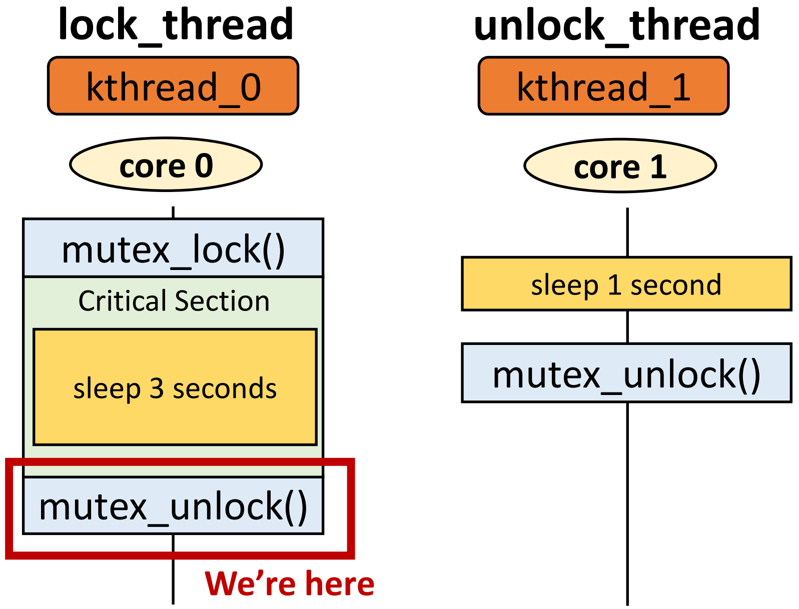


```
(gdb) p /x lock->owner
$3 = {
  counter = 0x0
}
(gdb) c
Continuing.
---------------------------[ STACK ]---
0x19efeb0:      Error while running hook_stop:
Cannot access memory at address 0x19efeb0

Thread 2 hit Breakpoint 2, __mutex_unlock_slowpath (lock=0xffffffffc0002000, ip=
<optimized out>) at /home/adrian/git-repo/gdb-linux-real-mode/src/linux-5.11/ker
nel/locking/mutex.c:1251
1251                    if (old == owner) {
(gdb) p /x old
$4 = 0x0
(gdb) p /x owner
$5 = 0x0
(gdb) p /x lock->owner
$6 = {
  counter = 0x0
}
```

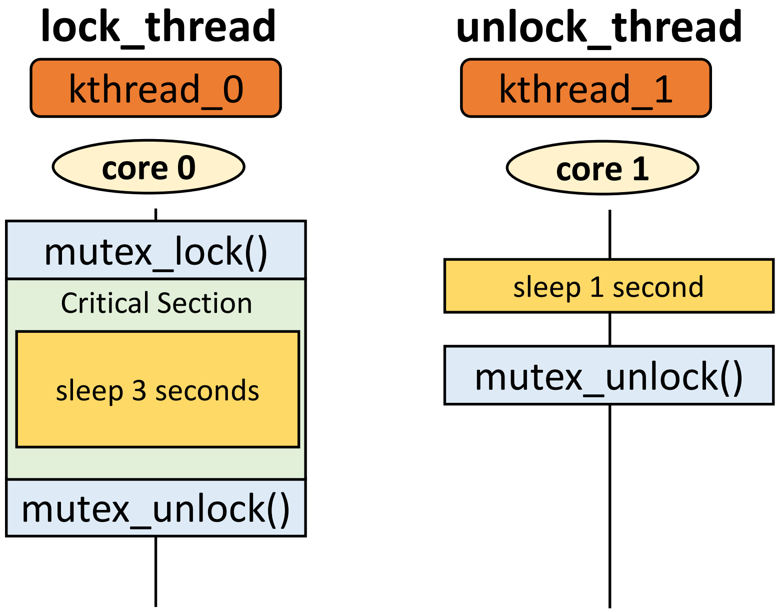
1 breakpoint

2 lock owner = old

3 lock->owner is set 0 by atomic_long_cmpxchg_release()



# mutex_unlock(): [lab] Behavior observation when lock owner != unlocker’s task (Mutex ownership)



**Takeaways**

1. [Fastpath] Linux kernel checks mutex’s ownership
2. [Slowpath] Linux kernel does not check mutex’s ownership when unlocking a mutex
   - ✓ Developers must take care of mutex_lock/mutex_unlock pair
   - ✓ Slowpath prints a warning message if mutex debug option is enabled.
   - ✓ Different from the concept: Only the lock owner has the permission to unlock the mutex

# mutex_unlock(): [lab] Behavior observation when lock owner != unlocker’s task (Mutex ownership)

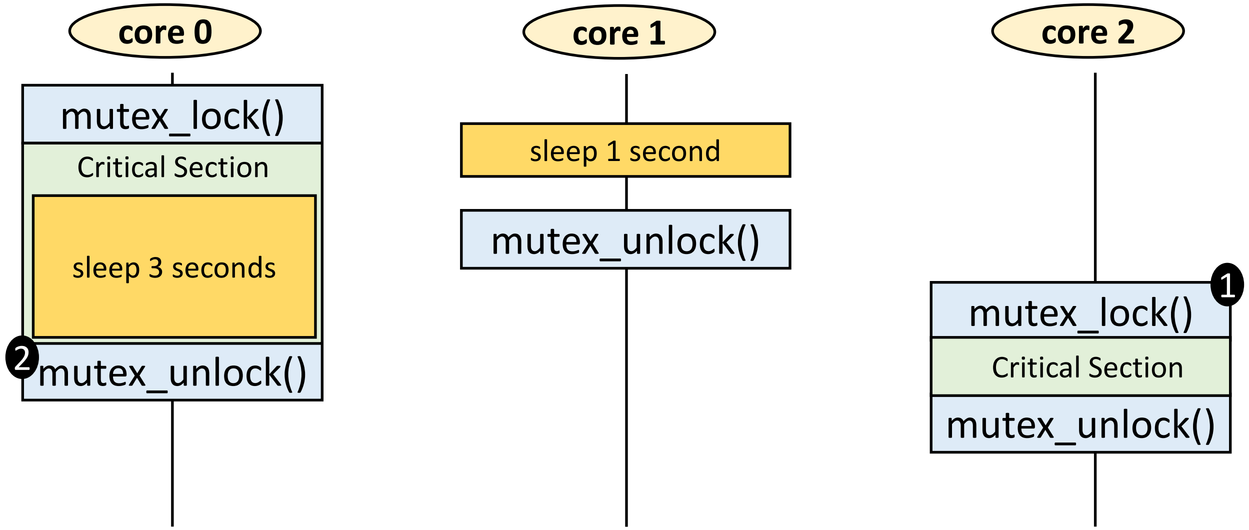
**Why doesn’t slowpath check ownership?**

1. Ownership checking is only for developers and not enforced by Linux kernel.
   - ✓ Developers need to take care of it.

**Quotes**

1. From **Generic Mutex Subsystem**
   - ✓ Mutex Semantics
     - Only one task can hold the mutex at a time.
     - Only the owner can unlock the mutex.
     - …
   - ✓ These semantics are fully enforced when CONFIG_DEBUG_MUTEXES is enabled

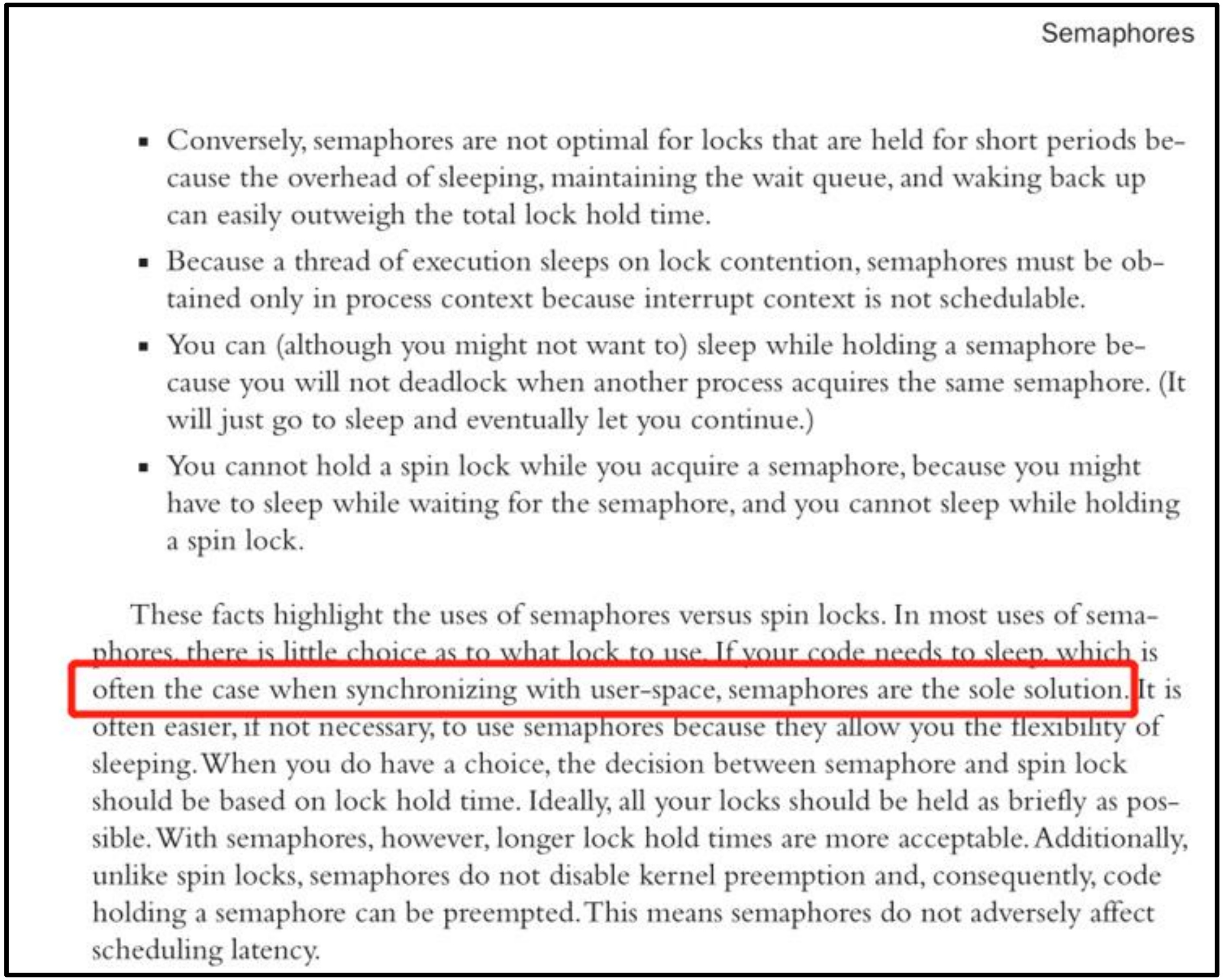
# Think about...



1 Will it acquire the mutex lock successfully?

2 Will this unlock the mutex lock acquired by core 2?

# Q&A #1: Semaphore can be used to synchronize with user-space



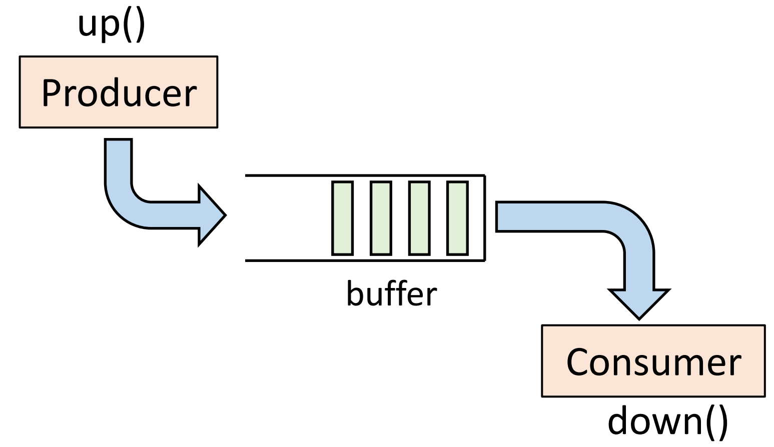
Semaphores

- Conversely, semaphores are not optimal for locks that are held for short periods because the overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time.
- Because a thread of execution sleeps on lock contention, semaphores must be obtained only in process context because interrupt context is not schedulable.
- You can (although you might not want to) sleep while holding a semaphore because you will not deadlock when another process acquires the same semaphore. (It will just go to sleep and eventually let you continue.)
- You cannot hold a spin lock while you acquire a semaphore, because you might have to sleep while waiting for the semaphore, and you cannot sleep while holding a spin lock.

These facts highlight the uses of semaphores versus spin locks. In most uses of semaphores, there is little choice as to what lock to use. If your code needs to sleep, which is often the case when synchronizing with user-space, semaphores are the sole solution. It is often easier, if not necessary, to use semaphores because they allow you the flexibility of sleeping. When you do have a choice, the decision between semaphore and spin lock should be based on lock hold time. Ideally, all your locks should be held as briefly as possible. With semaphores, however, longer lock hold times are more acceptable. Additionally, unlike spin locks, semaphores do not disable kernel preemption and, consequently, code holding a semaphore can be preempted. This means semaphores do not adversely affect scheduling latency.

* Screenshot captured from: Chapter 10, Linux Kernel Development, 3rd Edition, Robert Love

## [Semaphore] Producer/Consumer Concept

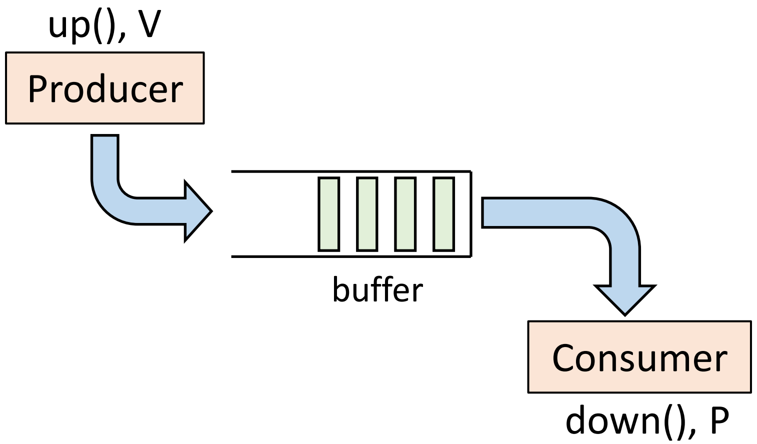


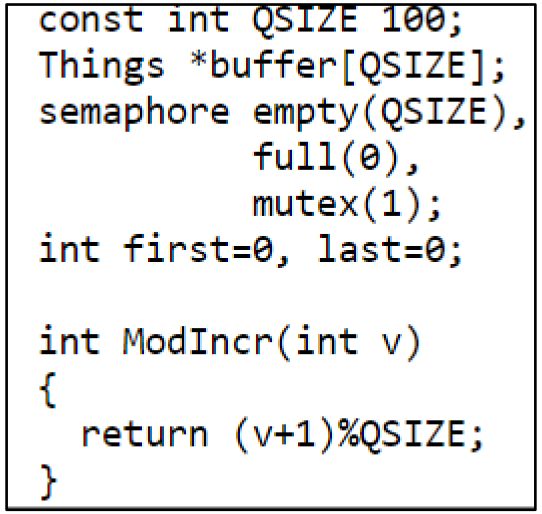
**Principle**

- Consumer waits if buffer is empty
- Producer waits if buffer is full
- Only one process can manipulate the buffer at a time (mutual exclusion)

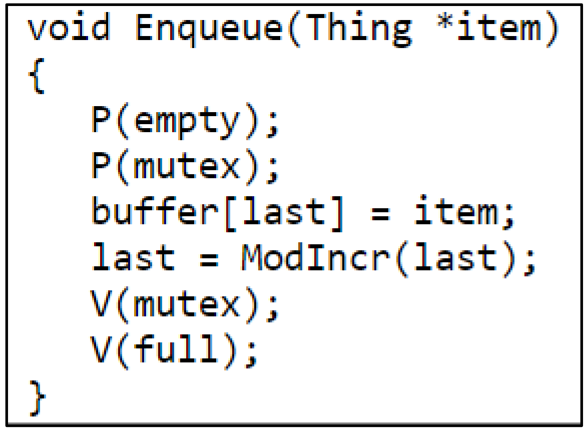
# Q&A #1: Semaphore can be used to synchronize with user-space

## [Semaphore] Producer/Consumer Concept

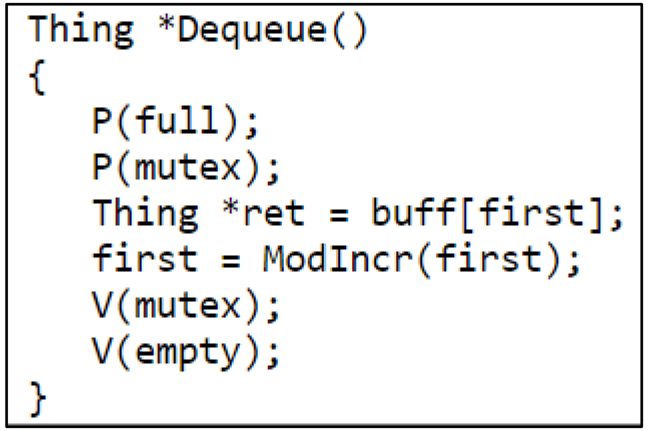




```
const int QSIZE 100;
Things *buffer[QSIZE];
semaphore empty(QSIZE),
          full(0),
          mutex(1);
int first=0, last=0;

int ModIncr(int v)
{
  return (v+1)%QSIZE;
}
```



```
void Enqueue(Thing *item)
{
   P(empty);
   P(mutex);
   buffer[last] = item;
   last = ModIncr(last);
   V(mutex);
   V(full);
}
```

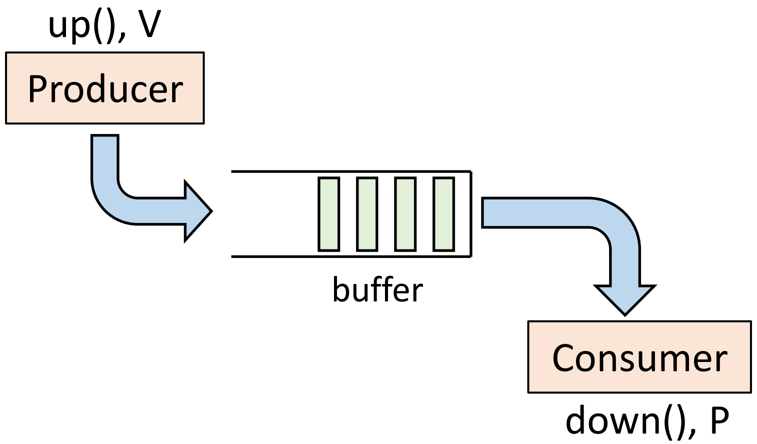


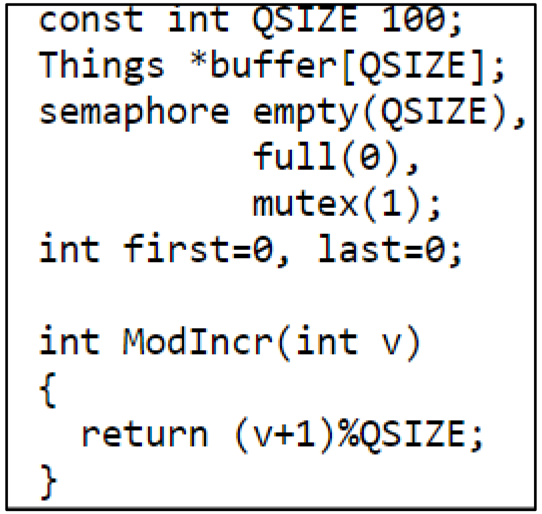
```
Thing *Dequeue()
{
   P(full);
   P(mutex);
   Thing *ret = buff[first];
   first = ModIncr(first);
   V(mutex);
   V(empty);
}
```

Code reference: CS 537 Notes, Section #6: Semaphores and Producer/Consumer Problem

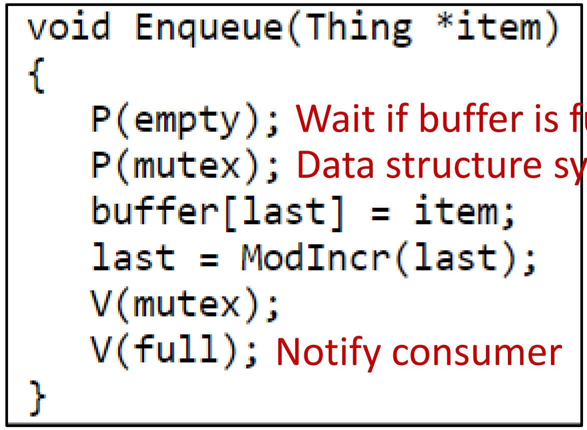
# Q&A #1: Semaphore can be used to synchronize with user-space

## [Semaphore] Producer/Consumer Concept

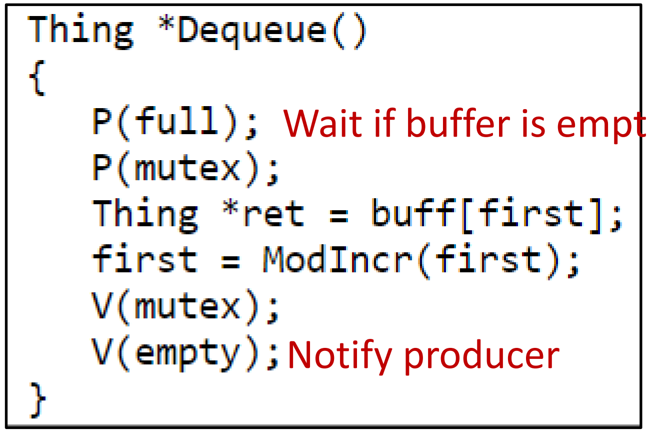




```
const int QSIZE 100;
Things *buffer[QSIZE];
semaphore empty(QSIZE),
          full(0),
          mutex(1);
int first=0, last=0;

int ModIncr(int v)
{
  return (v+1)%QSIZE;
}
```



```
void Enqueue(Thing *item)
{
    P(empty); Wait if buffer is full
    P(mutex); Data structure synchronization
    buffer[last] = item;
    last = ModIncr(last);
    V(mutex);
    V(full); Notify consumer
}
```

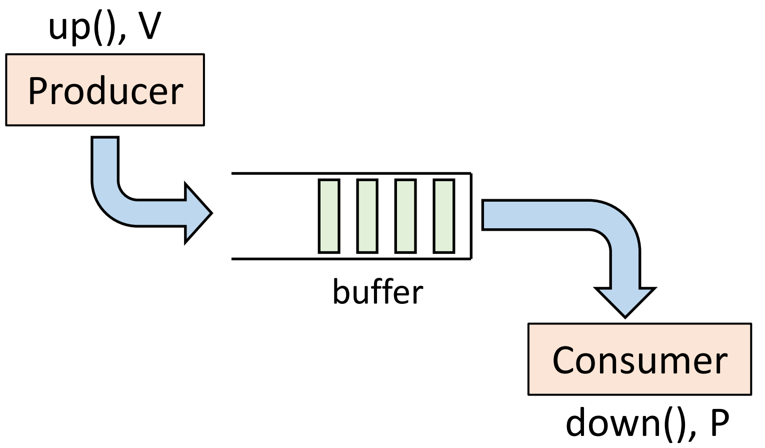


```
Thing *Dequeue()
{
    P(full);  Wait if buffer is empty
    P(mutex);
    Thing *ret = buff[first];
    first = ModIncr(first);
    V(mutex);
    V(empty);Notify producer
}
```

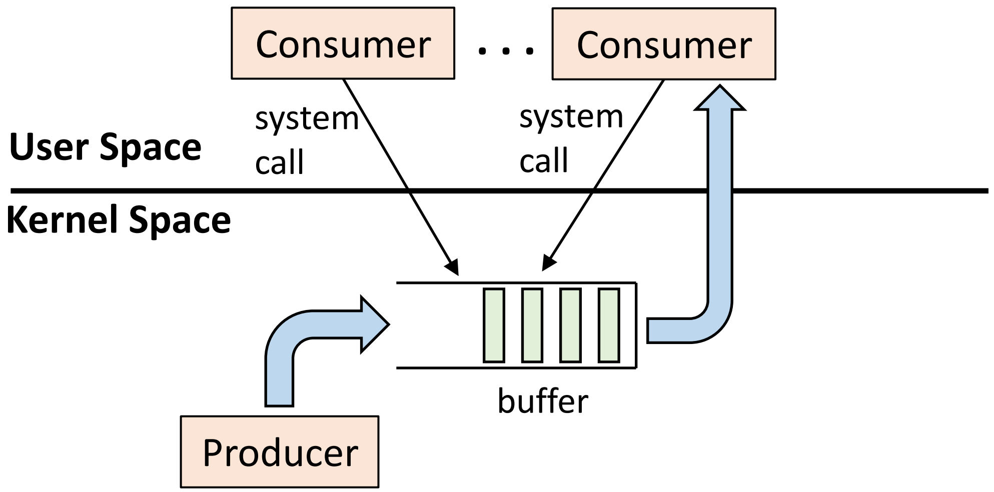
Code reference: CS 537 Notes, Section #6: Semaphores and Producer/Consumer Problem

# Q&A #1: Semaphore can be used to synchronize with user-space

## [Semaphore] Producer/Consumer Concept



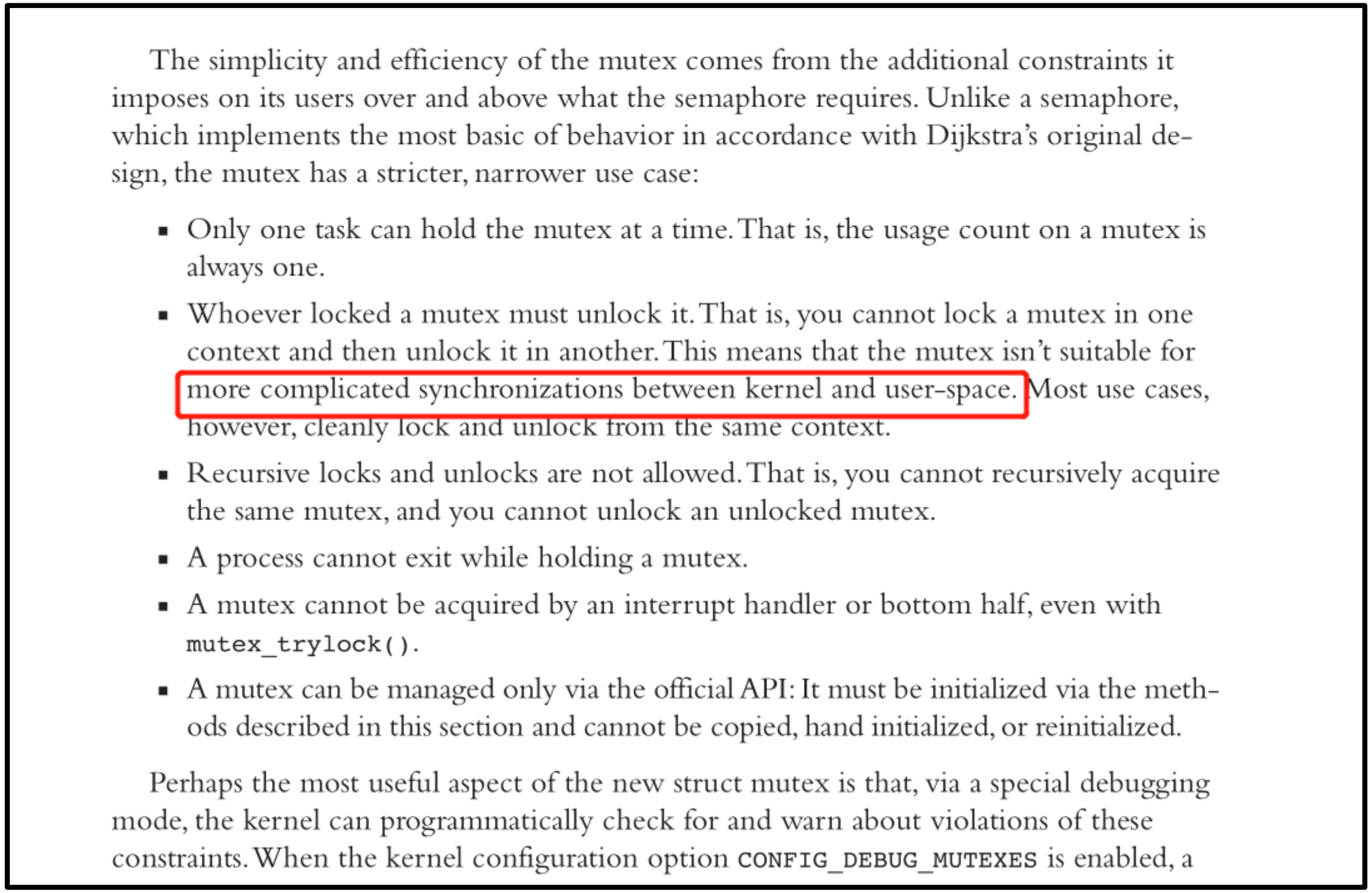
## Possible Scenario



**Note**

1. up()/down() invocations are done in kernel.

# Q&A #2: Mutex isn't suitable for synchronizations between kernel and user-space



The simplicity and efficiency of the mutex comes from the additional constraints it imposes on its users over and above what the semaphore requires. Unlike a semaphore, which implements the most basic of behavior in accordance with Dijkstra's original design, the mutex has a stricter, narrower use case:

- Only one task can hold the mutex at a time. That is, the usage count on a mutex is always one.
- Whoever locked a mutex must unlock it. That is, you cannot lock a mutex in one context and then unlock it in another. This means that the mutex isn't suitable for more complicated synchronizations between kernel and user-space. Most use cases, however, cleanly lock and unlock from the same context.
- Recursive locks and unlocks are not allowed. That is, you cannot recursively acquire the same mutex, and you cannot unlock an unlocked mutex.
- A process cannot exit while holding a mutex.
- A mutex cannot be acquired by an interrupt handler or bottom half, even with `mutex_trylock()`.
- A mutex can be managed only via the official API: It must be initialized via the methods described in this section and cannot be copied, hand initialized, or reinitialized.

Perhaps the most useful aspect of the new struct mutex is that, via a special debugging mode, the kernel can programmatically check for and warn about violations of these constraints. When the kernel configuration option `CONFIG_DEBUG_MUTEXES` is enabled, a

**Explanation**

1. [Mutex] ownership!
   - ✓ Whoever locked a mutex must unlock it

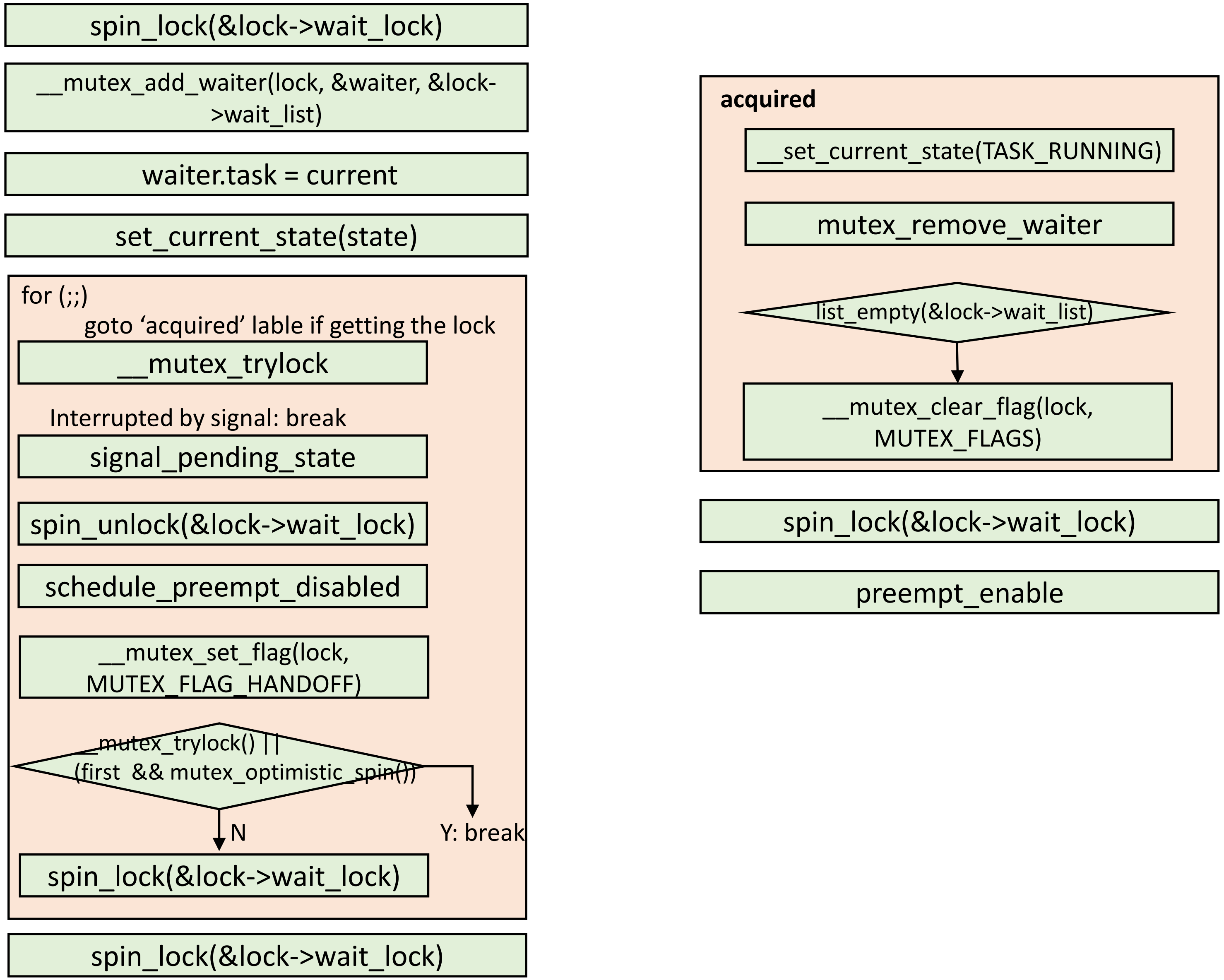
* Screenshot captured from: Chapter 10, Linux Kernel Development, 3rd Edition, Robert Love

# Reference

- Generic Mutex Subsystem
- Wound/Wait Deadlock-Proof Mutex Design
- Mutexes and Semaphores Demystified
- MCS locks and qspinlocks
- Linux中的mutex机制[一] - 加锁和osq lock

# Backup

# mutex_lock(): slowpath



spin_lock(&lock->wait_lock)

__mutex_add_waiter(lock, &waiter, &lock->wait_list)

waiter.task = current

set_current_state(state)

for (;;)

goto 'acquired' lable if getting the lock

__mutex_trylock

Interrupted by signal: break

signal_pending_state

spin_unlock(&lock->wait_lock)

schedule_preempt_disabled

__mutex_set_flag(lock, MUTEX_FLAG_HANDOFF)

__mutex_trylock() || (first && mutex_optimistic_spin())

N

Y: break

spin_lock(&lock->wait_lock)

spin_lock(&lock->wait_lock)

**acquired**

__set_current_state(TASK_RUNNING)

mutex_remove_waiter

list_empty(&lock->wait_list)

__mutex_clear_flag(lock, MUTEX_FLAGS)

spin_lock(&lock->wait_lock)

preempt_enable