Locking

Son Ju Hyung

tooson9010@gmail.com

Locking

- x86 기준 다양한 API 중 기본적인 lock & unlock mechanism
- Included
  - atomic operations
  - spinlocks(normal spinlock, ticket spinlock, queue-based spinlock)
  - rwspinlock(normal rwlock, queue-based rwlock)
  - semaphores(semaphore, rwsemaphore)
  - mutex(normal mutex, real time mutex, wait-wound mutex)
  - reader/writer semaphores
  - sequential locks
  - completion variables
  - big kernel locks
  - read-copy-update
  - …

Atomic Operations

- 공유 변수의 증감 시, load from memory ➔ increment/decrement ➔ store to memory 의 과정과 같은 여러 명령이 순차적으로 수행되는 것을 보장하는 방법
- 시스템에서 사용할 수 있는 가장 작은 단위의 동기화 기법 이며 deadlock 이 없고, 동기화 기법 중 가장 빠른 성능을 가지고 있다.

```
typedef struct {
    int counter;
} atomic_t;

#ifdef CONFIG_64BIT
typedef struct {
    long counter;
} atomic64_t;
#endif
```

LOCK_PREFIX ➔ lock

- memory bus 에 lock signal 을 주어 CPU 들간 shared memory 영역에 대한 잠금 수행
- single read-modify-write instruction(inc, xchg, add …) 에 대해 atomic 보장

linux/arch/x86/include/asm/atomic.h

```
static __always_inline void atomic_add(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "addl %1,%0"
             : "+m" (v->counter)
             : "ir" (i));
}

static __always_inline void atomic_sub(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "subl %1,%0"
             : "+m" (v->counter)
             : "ir" (i));
}

static __always_inline void atomic_inc(atomic_t *v)
{
    asm volatile(LOCK_PREFIX "incl %0"
             : "+m" (v->counter));
}

static __always_inline void atomic_dec(atomic_t *v)
{
    asm volatile(LOCK_PREFIX "decl %0"
             : "+m" (v->counter));
}
```

…

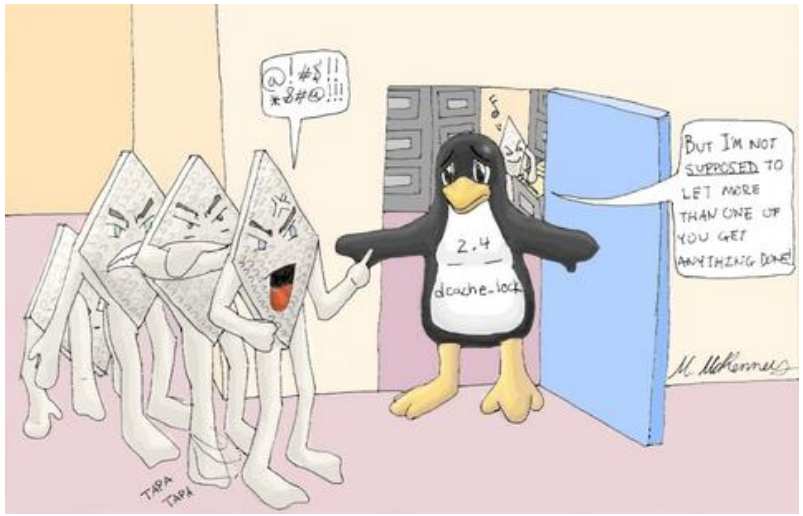
linux/arch/x86/include/asm/cmpxchg.h

```
#define __xchg_op(ptr, arg, op, lock)                   \
    ({                                                  \
            __typeof__ (*(ptr)) __ret = (arg);          \
        switch (sizeof(*(ptr))) {                       \
        case __X86_CASE_B:                              \
            asm volatile (lock #op "b %b0, %1\n"        \
                          : "+q" (__ret), "+m" (*(ptr)) \
                          : : "memory", "cc");          \
            break;                                      \
        case __X86_CASE_W:                              \
            asm volatile (lock #op "w %w0, %1\n"        \
                          : "+r" (__ret), "+m" (*(ptr)) \
                          : : "memory", "cc");          \
            break;                                      \
        case __X86_CASE_L:                              \
            asm volatile (lock #op "l %0, %1\n"         \
                          : "+r" (__ret), "+m" (*(ptr)) \
                          : : "memory", "cc");          \
            break;                                      \
        case __X86_CASE_Q:                              \
            asm volatile (lock #op "q %q0, %1\n"        \
                          : "+r" (__ret), "+m" (*(ptr)) \
                          : : "memory", "cc");          \
            break;                                      \
        default:                                        \
            __ ## op ## _wrong_size();                  \
        }                                               \
        __ret;                                          \
    })
```

…

Spinlock

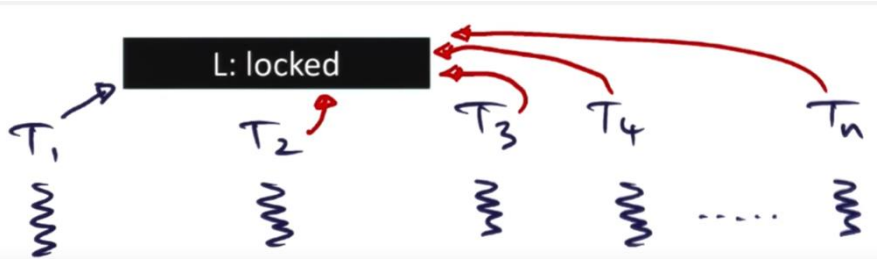
- critical section 에 동시에 여러 process 의 접근을 막기 위한 mechanism 으로 lock 을 잡지 못한 process 는 lock 이 풀릴 때 까지 대기.
- critical section 의 수행이 짧은 경우에 적합하며, lock 을 잡을 상태로 sleep/선점 불가능.
- interrupt service routine 에서 사용될 경우, local interrupt 를 disable 해주어야 함.
- semaphore/mutex/seqlock 등 다른 lock 에서 wait list 등을 보호하기 위하여 사용.
- kernel 내의 사용 용도
  - 굉장히… 엄청 많은 곳에서 사용…



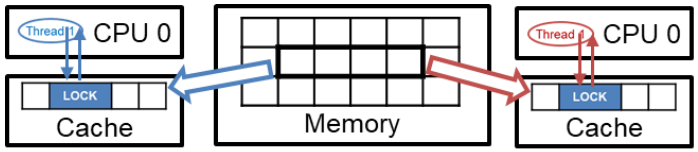

```
int lock(lock)
{
    while (test_and_set(lock) == 1)
        ;
    return 0;
}

int unlock(lock)
{
    lock=0;

    return lock;
}
```

Spinlock and Patches

- ***normal spinlock***
  - 기본 동작은 test_and_set 기반으로 atomic operations 을 활용한 lock 변수 조작
    - Problem1 : lock 이 풀릴 때, lock 을 받는 순서 보장 못하며 contention 발생 (lock 요청을 먼저한 thread 가 나중에 lock 요청을 한 thread 보다 늦게 lock 을 받을 수 있음) ➔ unfairness



    - Problem2 :단일 lock 변수에 대한 접근 ➔ cache line invalidation/bouncing 자주 발생

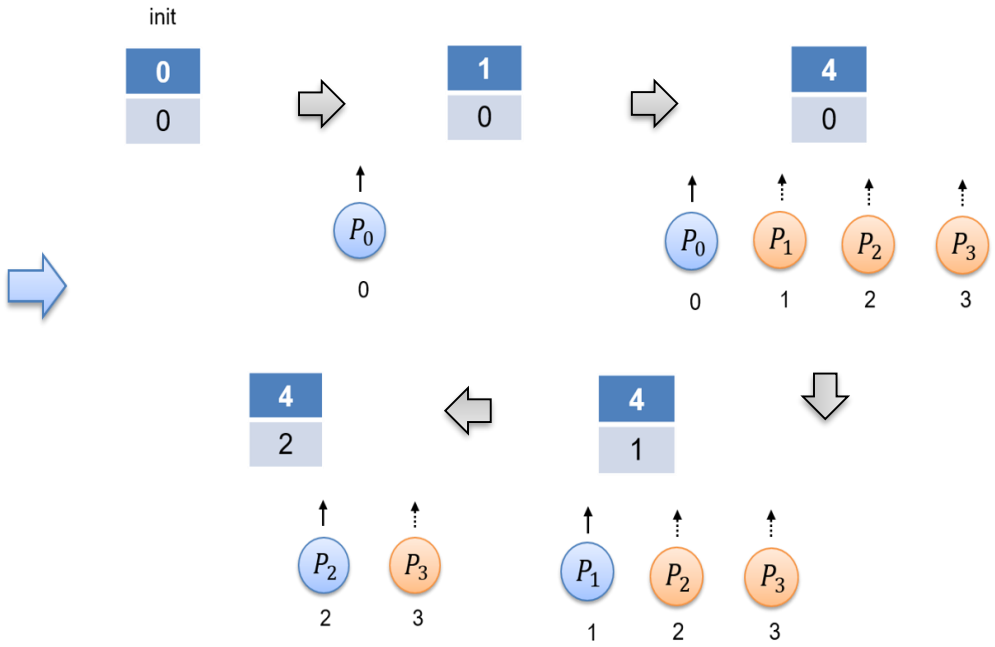


- ***ticket lock***
  - patch from 2.6.5
  - lock 획득의 순서 보장을 통해 Problem 1 개선
- ***queued lock***
  - patch from 4.2
  - cpu 별 local variable 에서의 spinning 을 통해 Problem 2 개선
  - x86에서 ticket lock code 제거 및 queue lock 으로 완전 대체(v4.9)

Ticket Spinlocks

- Linux Kernel v2.6.5 이후 patch 된 lock 으로 normal spinlock에서 lock waiter 들이 공평하게 lock 을 얻을 수 있도록 하여 unlock 시 발생하는 lock contention 줄임
- lock 요청 시, next ticket counter 증가시키며, ticket number 증가 및 자신의 ticket 과 비교하며 spinning, lock 해제 시, 현재 serving 되는 lock counter 증가
- raw_spinlock 에서 arch_spinlock_t 는 CONFIG_QUEUED_SPINLOCKS 설정에 따라 달라졌지만 현재 x86_64 에서는 v4.9 이후부터 x86 specific arch_spinlock_t 가 아닌 kernel 의 qspinlock 을 사용 (ticket spinlock 대용으로 queue spinlock 사용하도록 patch 됨 이후 2016년 5월 x86 에서 ticket lock code 제거 patch 됨)

```
struct lock {
        int next_ticket;
        int now_serving;
}
acquire_lock(L) :
    int my_ticket = F&I(L->next_tichet) ;
    Loop
        pause(myticket – L->now_serving) ;
        if( L-> now_serving == my_ticket )
            return ;
release_lock(L) :
    L -> now_serving ++;
```



Ticket Spinlocks

- data structures

linux/include/linux/spinlock_types.h

```
typedef struct spinlock {
    union {
        struct raw_spinlock rlock;
        // normal spin lock
#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
        struct {
            u8 __padding[LOCK_PADSIZE];
            struct lockdep_map dep_map;
        };
#endif
    };
} spinlock_t;
```

```
typedef struct raw_spinlock {
    arch_spinlock_t raw_lock;
    // arcitecture-specific spinlock
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
    // 특정 processor 가 lock 을
    // 잡고 있는 상태에서 다른
    // processor 가 lock 을 대기중일
    // 경우 1로 설정됨
    // (long time locking 방지용도)
#endif
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} raw_spinlock_t;
```

linux/include/linux/spinlock_types_up.h

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

include/asm-generic/qspinlock_types.h

```
typedef struct qspinlock {
    atomic_t    val;
    /*
     * Bitfields in the atomic value:
     *
     * When NR_CPUS < 16K
     *  0- 7: locked byte
     *     8: pending
     *  9-15: not used
     * 16-17: tail index -> lock 이 잡힌 context.. 4 개중 하나
     * 18-31: tail cpu (+1) -> 현재 lock 잡고있는 cpu number
     *                         14 bit 크기 이므로 16384 개의
     *                         core 까지 표현 가능
     *
     * When NR_CPUS >= 16K
     *  0- 7: locked byte
     *     8: pending
     *  9-10: tail index
     * 11-31: tail cpu (+1)
     */
} arch_spinlock_t;
```

v4.8

linux/arch/x86/include/asm/spinlock_types.h

```
typedef struct arch_spinlock {
    union {
        __ticketpair_t head_tail;
        // lock counter 값으로 두가지로 구성
        struct __raw_tickets {
            __ticket_t head, tail;
            // head : 현재 lock 을 잡은 thread 가
            //        수행중인 index
            // tail : lock 을 잡으려는 다음 thread
            //        에게 줄 index
        } tickets;
    };
} arch_spinlock_t;
```

```
typedef u8  __ticket_t;
typedef u16 __ticketpair_t;
#else
typedef u16 __ticket_t;
typedef u32 __ticketpair_t;
```

CPU 개수	head	tail
• < 256 개	8	8
• > 256 개	16	16

- initialization
  - 초기 head, tail 을 모두 0으로 설정

```
#define spin_lock_init(_lock)                   \
do {                                    \
    spinlock_check(_lock);              \
    raw_spin_lock_init(&(_lock)->rlock);        \
} while (0)

# define raw_spin_lock_init(lock)               \
    do { *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock); } while (0)
    /* spinlock 을 released 상태인 0 상태로 초기화 */
```

```
#define __RAW_SPIN_LOCK_INITIALIZER(lockname)   \
    {                                   \
    .raw_lock = __ARCH_SPIN_LOCK_UNLOCKED,  \
    /* unlocked 로 설정 */ \
    SPIN_DEBUG_INIT(lockname)           \
    /* lockdep 관련 디버깅 필드 초기화 */ \
    SPIN_DEP_MAP_INIT(lockname) }

#define __ARCH_SPIN_LOCK_UNLOCKED   { { 0 } }
```

Ticket Spinlocks

- spin_lock (v4.8 기준)
  - next ticket 증가 값을 설정

```
#ifdef CONFIG_PARAVIRT_SPINLOCKS
#define __TICKET_LOCK_INC    2
#define TICKET_SLOWPATH_FLAG    ((__ticket_t)1)
#else
#define __TICKET_LOCK_INC    1
#define TICKET_SLOWPATH_FLAG    ((__ticket_t)0)
#endif
```

  - xadd 를 통해 증가 값을 global ticket lock 변수의 tail 값에 더하고, 옛날 값을 가져옴

    ➔ ticket 번호 증가 및 자신의 ticket 번호 가져옴
  - 주기적으로 head 값을 inc.head 에 읽어 들여 자신의 ticket 번호(inc.tail) 와 같은지 검사
    - inc.head == inc.tail
      - 자신이 lock 받을 차례
    - inc.head < inc.tail
      - 차례 아니므로 spinning
      - spu_relax

```
static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}
```

```
static __always_inline void arch_spin_lock(arch_spinlock_t *lock)
{
    register struct __raw_tickets inc = { .tail = TICKET_LOCK_INC };
    // register 변수를 통해 local cpu 에 생성시도
    // 한번에 증가되는 lock counter 값 초기화

    inc = xadd(&lock->tickets, inc);
    // lock->tickets 에 inc 를 더함. 즉 next ticket 번호를 증가하고
    // 자신의 ticket 번호를 가져옴
    if (likely(inc.head == inc.tail))
        goto out;
    // 현재 자신의 ticket 번호가 현재 serve 가능한 ticket 번호(head) 와
    // 일치하는지 검사 후, 같다면 lock 을 바로 얻을 수 있으므로 out
    for (;;) {
        unsigned count = SPIN_THRESHOLD; // 1 << 15

        do {
            inc.head = READ_ONCE(lock->tickets.head);
            // global 로부터 head 를 다시 가져와 inc 의
            // head 정보 update (lock ticket serving status)
            if (__tickets_equal(inc.head, inc.tail))
                goto clear_slowpath;
                // 현재 가진 ticket 차례인지 검사
            cpu_relax();
        } while (--count);
        __ticket_lock_spinning(lock, inc.tail);
    }
clear_slowpath:
    __ticket_check_and_clear_slowpath(lock, inc.head);
out:
    barrier();  /* make sure nothing creeps before the lock is taken */
}
```

Ticket Spinlocks

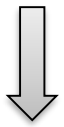
- spin_lock (v4.8 기준)
  - next ticket 증가 값을 설정

```
#ifdef CONFIG_PARAVIRT_SPINLOCKS
#define __TICKET_LOCK_INC    2
#define TICKET_SLOWPATH_FLAG    ((__ticket_t)1)
#else
#define __TICKET_LOCK_INC    1
#define TICKET_SLOWPATH_FLAG    ((__ticket_t)0)
#endif
```

  - xadd 를 통해 증가 값을 global ticket lock 변수의 tail 값에 더하고, 옛날 값을 가져옴

    ➔ ticket 번호 증가 및 자신의 ticket 번호 가져옴
  - 주기적으로 head 값을 inc.head 에 읽어 들여 자신의 ticket 번호(inc.tail) 와 같은지 검사
    - inc.head == inc.tail
      - 자신이 lock 받을 차례
    - inc.head < inc.tail
      - 차례 아니므로 spinning
      - spu_relax
- spin_unlock (v4.8 기준)
  - ticket 번호를 증가시켜(head) lock 을 기다리는 waiter 가 lock 을 잡을 수 있도록 해줌

```
static __always_inline void spin_unlock(spinlock_t *lock)
{
    raw_spin_unlock(&lock->rlock);
}
```



```
static __always_inline void arch_spin_unlock(arch_spinlock_t *lock)
{
    if (TICKET_SLOWPATH_FLAG &&
        static_key_false(&paravirt_ticketlocks_enabled)) {
        __ticket_t head;

        BUILD_BUG_ON(((__ticket_t)NR_CPUS) != NR_CPUS);

        head = xadd(&lock->tickets.head, TICKET_LOCK_INC);

        if (unlikely(head & TICKET_SLOWPATH_FLAG)) {
            head &= ~TICKET_SLOWPATH_FLAG;
            __ticket_unlock_kick(lock, (head + TICKET_LOCK_INC));
        }
    } else
        __add(&lock->tickets.head, TICKET_LOCK_INC, UNLOCK_LOCK_PREFIX);
}
```

Ticket Spinlocks

- spin_lock (v4.8 기준)
  - next ticket 증가 값을 설정

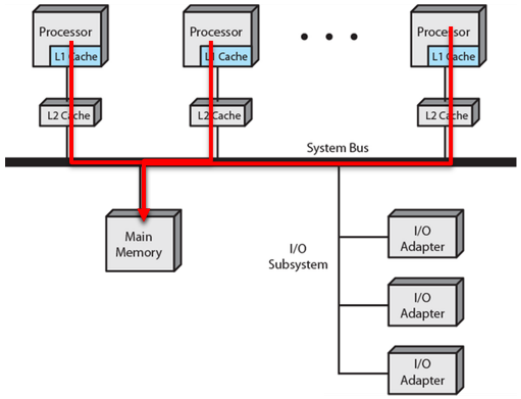
```
#ifdef CONFIG_PARAVIRT_SPINLOCKS
#define __TICKET_LOCK_INC    2
#define TICKET_SLOWPATH_FLAG    ((__ticket_t)1)
#else
#define __TICKET_LOCK_INC    1
#define TICKET_SLOWPATH_FLAG    ((__ticket_t)0)
#endif
```

  - xadd 를 통해 증가 값을 global ticket lock 변수의 tail 값에 더하고, 옛날 값을 가져옴

    ➔ ticket 번호 증가 및 자신의 ticket 번호 가져옴
  - 주기적으로 head 값을 inc.head 에 읽어 들여 자신의 ticket 번호(inc.tail) 와 같은지 검사
    - inc.head == inc.tail
      - 자신이 lock 받을 차례
    - inc.head < inc.tail
      - 차례 아니므로 spinning
      - spu_relax
- spin_unlock (v4.8 기준)
  - ticket 번호를 증가시켜(head) lock 을 기다리는 waiter 가 lock 을 잡을 수 있도록 해줌
- ticket lock 장단점은?
  - **lock 을 잡을 순서 지정**
  - **cache line invalidation 은 아직 가능**

```
static __always_inline void spin_unlock(spinlock_t *lock)
{
    raw_spin_unlock(&lock->rlock);
}
```

```
static __always_inline void arch_spin_unlock(arch_spinlock_t *lock)
{
    if (TICKET_SLOWPATH_FLAG &&
        static_key_false(&paravirt_ticketlocks_enabled)) {
        __ticket_t head;

        BUILD_BUG_ON(((__ticket_t)NR_CPUS) != NR_CPUS);

        head = xadd(&lock->tickets.head, TICKET_LOCK_INC);

        if (unlikely(head & TICKET_SLOWPATH_FLAG)) {
            head &= ~TICKET_SLOWPATH_FLAG;
            __ticket_unlock_kick(lock, (head + TICKET_LOCK_INC));
        }
    } else
        __add(&lock->tickets.head, TICKET_LOCK_INC, UNLOCK_LOCK_PREFIX);
}
```



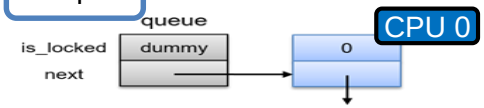
Queued-based Spinlocks

- MCS lock 기반 알고리즘으로 동작
  - 공유되는 하나의 lock 변수가 아닌 각각의 CPU local 변수에 대해 spinning 하며 lock 을 얻으려 시도하는 lock 으로 cache line invalidation 을 줄일 수 있음
  - lock lock 변수가 1 이라면 spinning 하며 대기, 0 이라면 spinning 풀려나 lock 잡음

Step 1

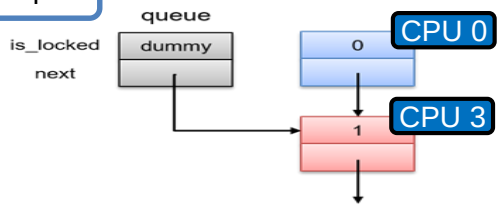
1. lock 이 잡혀있지 않은 상태

Step 2



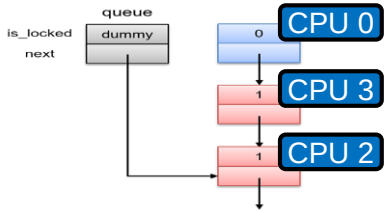
1. core 0 에서 수행되는 thread 가 lock 획득 시도
2. queue 의 next 가 null 이었으므로 성공

Step 3



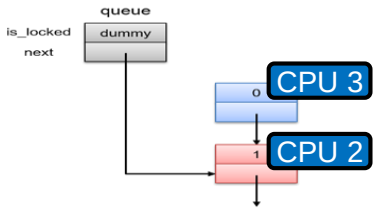
1. core 3 에서 수행되는 thread 가 lock 획득 시도 및 queue->next 가 null 아니므로 획득 실패 및 local lock 변수 1 로 설정
2. next 를 core 3 local 주소로 설정하여 연결
3. lock 변수 보고 spinning 하며 lock 대기

Step 4



1. core 2 lock 요청에 대해 lock 획득 실패
2. next 를 core 2 의 local 주소로 설정하여 연결
…

Step 5



1. core 0 에서 수행되던 thread 는 unlock 하며 next 의 lock 상태를 0 으로 변경
2. core 3 에서 spinning 하던 thread 가 lock 획득

Queued-based Spinlocks

- MCS lock 기반 알고리즘으로 동작
  - 공유되는 하나의 lock 변수가 아닌 각각의 CPU local 변수에 대해 spinning 하며 lock 을 얻으려 시도하는 lock 으로 cache line invalidation 을 줄일 수 있음
  - lock lock 변수가 1 이라면 spinning 하며 대기, 0 이라면 spinning 풀려나 lock 잡음

```
void lock(...)
{
    lock.next = NULL;
    ancestor = put_lock_to_queue_and_return_ancestor(queue, lock);

    // if we have ancestor, the lock already acquired and we
    // need to wait until it will be released
    if (ancestor)
    {
        lock.locked = 1;
        ancestor.next = lock;

        while (lock.is_locked == true)
            ;
    }

    // in other way we are owner of the lock and may exit
}
```

```
void unlock(...)
{
    // do we need to notify somebody or we are alonw in the
    // queue?
    if (lock.next != NULL) {
        // the while loop from the lock() function will be
        // finished
        lock.next.is_locked = false;
        // delete ourself from the queue and exit
        ...
        ...
        ...
        return;
    }

    // So, we have no next threads in the queue to notify about
    // lock releasing event. Let's just put `0` to the lock, will
    // delete ourself from the queue and exit.
}
```

Queued-based Spinlocks

- MCS lock code 가 kernel 에 도입 (v3.15)
  - kernel 내부의 다양한 곳에 사용되는 32/64-byte word 크기 spinlock 변수에 맞추지 못하여 spinlock 대체 못하고 있었음(mcs_spin_lock 관련 api 있지만 사용 되진 않음)
- 32byte atomic_t spinlock 변수내의 각 bit 에 여러 정보를 embedded 한 4-byte queued spinlock 이 patch 됨
  - MCS lock array (normal, s/w interrupt, h/w interrupt, non-maskable interrupt) 을 미리 address 를 알고 있는 per-CPU variable 를 통해 접근
- v4.9 부터 기존 ticket spinlock 을 대체하여 queued spinlock 을 사용하도록 patch 됨(x86) 즉 x86_64 은 v4.9 이후부터 ticket lock 기반의 arch_* 이 queued_* 로 대체
  - linux/arch/x86/include/asm/spinlock.h ➔ include/asm-generic/qspinlock.h
  - default 로 ARCH_USE_QUEUED_SPINLOCK 설정됨 ➔ CONFIG_QUEUED_SPINLOCKS 자동 설정
  - 2016 년 5월 x86에서 ticket lock code 제거 patch 됨

linux/kernel/Kconfig.locks

```
config ARCH_USE_QUEUED_SPINLOCKS
    bool

config QUEUED_SPINLOCKS
    def_bool y if ARCH_USE_QUEUED_SPINLOCKS
    depends on SMP
```

x86 default

linux/arch/x86/Kconfig

```
#
# Arch settings
#
# ( Note that options that are marked 'if X86_64' could in principle be
#   ported to 32-bit as well. )
#
config X86
    def_bool y
    #
    # Note: keep this list sorted alphabetically
    #
    ...
    select ARCH_USE_QUEUED_SPINLOCKS
    ...
```

Queued-based Spinlocks

- MCS lock 알고리즘 과 함께 몇가지 optimization 도입
  - lock 을 잡고 있는 thread 가 하나도 없을 때….
    - mcs data structure 생성 없이 바로 lock 잡음
  - lock 을 잡으려는 thread (CPU) 가 두개 뿐일 때…
    - mcs data structure 생성 없이 lock 변수에 대하여 spinning (pending bit)
    - unlock 을 받을 mcs data structure 받는다면 contention 상황이 CPU 두개일 경우에는 per-CPU 변수 가진 cache line 올릴 필요 없음(1 cache line miss 줄임)

- ticket spinlock 만큼 fair하며 single-thread 에서 ticket lock 과 비슷한 속도 제공하며 high contention situation(NUMA, large core system) 에서 훨씬 빠름
  - AIM7 disk workload 수행 결과 vfs layer 와, ext4 file system code 내에서 116% 성능 향상

- 본 자료에서는 virtualization 상황(paravirtualized)은 다루지 않음
  - pvqspinlock 제외

Ticket Spinlocks

- data structures

linux/include/linux/spinlock_types.h

```
typedef struct spinlock {
    union {
        struct raw_spinlock rlock;
        // normal spin lock
#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
        struct {
            u8 __padding[LOCK_PADSIZE];
            struct lockdep_map dep_map;
        };
#endif
    };
} spinlock_t;

typedef struct raw_spinlock {
    arch_spinlock_t raw_lock;
    // arcitecture-specific spinlock
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
    // 특정 processor 가 lock 을
    // 잡고 있는 상태에서 다른
    // processor 가 lock 을 대기중일
    // 경우 1로 설정됨
    // (long time locking 방지용도 )
#endif
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} raw_spinlock_t;
```

linux/include/linux/spinlock_types_up.h

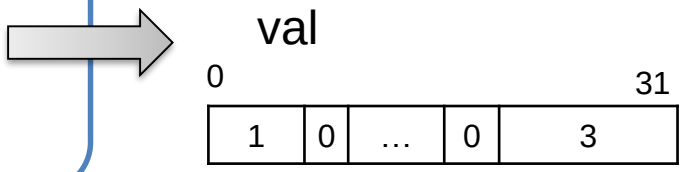
```
typedef struct {
    volatile unsigned int slock;
} arch_spinlock_t;
```

include/asm-generic/qspinlock_types.h

```
typedef struct qspinlock {
    atomic_t    val;
    /*
     * Bitfields in the atomic value:
     *
     * When NR_CPUS < 16K
     *  0- 7: locked byte
     *     8: pending
     *  9-15: not used
     * 16-17: tail index -> lock 이 잡힌 context.. 4 개중 하나
     * 18-31: tail cpu (+1) -> 현재 lock 잡고있는 cpu number
     *                         14 bit 크기 이므로 16384 개의
     *                         core 까지 표현 가능
     *
     * When NR_CPUS >= 16K
     *  0- 7: locked byte
     *     8: pending
     *  9-10: tail index
     * 11-31: tail cpu (+1)
     */
} arch_spinlock_t;
```

linux/arch/x86/include/asm/spinlock_types.h

```
typedef struct arch_spinlock {
    union {
        __ticketpair_t head_tail;
        // lock counter 값으로 두가지로 구성
        struct __raw_tickets {
            __ticket_t head, tail;
            // head : 현재 lock 을 잡은 thread 가
            //        수행중인 index
            // tail : lock 을 잡으려는 다음 thread
            //        에게 줄 index
        } tickets;
    };
} arch_spinlock_t;
```



- qspinlock
  - queued spinlock 은 lock 변수에 현재 lock 상태, lock 시도중인 마지막 CPU 번호 등의 정보 포함
    - 0-7 bit : 현재 lock 상태 (free or taken)
    - 8 bit : pending bit (contention 이 2 개일 경우 optimize)
    - 9-15 bit : 사용 안됨
    - 16-17 bit : 4 가지 MCS per-CPU mcs_spinlock array 들중 현재 사용되는 lock
    - 18-31 bit : MCS per-CPU lock queue 의 마지막 processor id (마지막에 lock 요청하여 대기중인 processor)

Ticket Spinlocks

- data structures

linux/include/linux/spinlock_types.h

```
typedef struct spinlock {
    union {
        struct raw_spinlock rlock;
        // normal spin lock
#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
        struct {
            u8 __padding[LOCK_PADSIZE];
            struct lockdep_map dep_map;
        };
#endif
    };
} spinlock_t;

typedef struct raw_spinlock {
    arch_spinlock_t raw_lock;
    // arcitecture-specific spinlock
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
    // 특정 processor 가 lock 을
    // 잡고 있는 상태에서 다른
    // processor 가 lock 을 대기중일
    // 경우 1로 설정됨
    // (long time locking 방지용도)
#endif
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} raw_spinlock_t;
```

linux/include/linux/spinlock_types_up.h

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

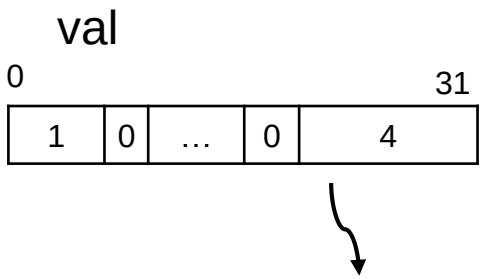
v4.9

include/asm-generic/qspinlock_types.h

```
typedef struct qspinlock {
    atomic_t    val;
    /*
     * Bitfields in the atomic value:
     *
     * When NR_CPUS < 16K
     *  0- 7: locked byte
     *     8: pending
     *  9-15: not used
     * 16-17: tail index -> lock 이 잡힌 context.. 4 개중 하나
     * 18-31: tail cpu (+1) -> 현재 lock 잡고있는 cpu number
     *                         14 bit 크기 이므로 16384 개의
     *                         core 까지 표현 가능
     *
     * When NR_CPUS >= 16K
     *  0- 7: locked byte
     *     8: pending
     *  9-10: tail index
     * 11-31: tail cpu (+1)
     */
} arch_spinlock_t;
```

linux/arch/x86/include/asm/spinlock_types.h

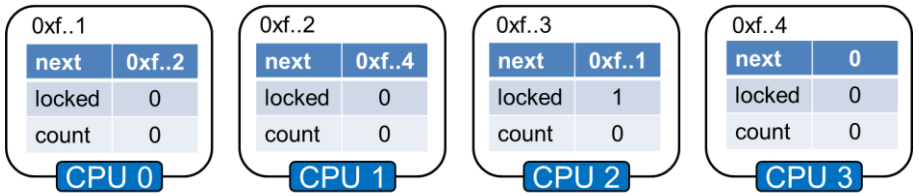
```
typedef struct arch_spinlock {
    union {
        __ticketpair_t head_tail;
        // lock counter 값으로 두가지로 구성
        struct __raw_tickets {
            __ticket_t head, tail;
            // head : 현재 lock 을 잡은 thread 가
            //        수행중인 index
            // tail : lock 을 잡으려는 다음 thread
            //        에게 줄 index
        } tickets;
    };
} arch_spinlock_t;
```



- mcs_spinlock
  - per-CPU variable 을 통해 MCS 알고리즘 구현

```
static DEFINE_PER_CPU_ALIGNED(struct mcs_spinlock, mcs_nodes[MAX_NODES]);
struct mcs_spinlock {
    struct mcs_spinlock *next;
    // queue 에서 lock 을 기다리는 next thread 의
    // mcs_spinlock 을 가리킴
    int locked; /* 1 if lock acquired */
    // 현재 thread 의 lock 에 대한 상태
    //  1 이면 : thread 가 lock 잡음
    //  0 이면 : 잡으려고 대기중
    int count;  /* nesting count, see qspinlock.c */
    // nested lock
    //  - normal task context
    //  - hardware interrupt context
    //  - software interrupt context
    //  - non-maskable interrupt context
};
```

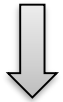
CPU 2 ➔ CPU 0 ➔ CPU 1 ➔ CPU 4



Queued-based Spinlocks

- spin_lock
  - spin_lock 수행 전, 선점 비활성화, lockdep 관련 debugging 정보 설정 등 수행
  - x86 의 경우, 기존 architecture dependent 한 spinlock 구현인 arch_spin_lock 이 queue_spin_lock 으로 연결

```
static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}

#define raw_spin_lock(lock) _raw_spin_lock(lock)
#define _raw_spin_lock(lock) __raw_spin_lock(lock)

static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    // 선점 비활성화, unlock 시 lock release 하고 재 활성화 됨
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    // interrupt state 저장 및 비활성화 후, lock 잡기 전에
    // runtime lock validator 수행
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
    // 진짜로 spinlock 을 얻는 함수
}
```



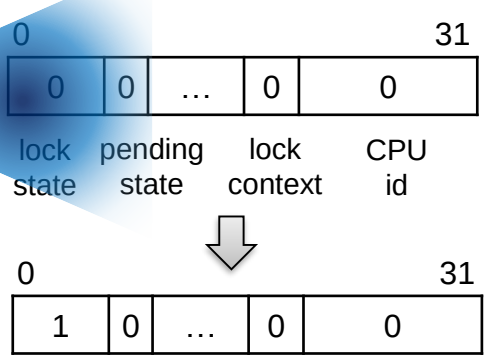
```
void do_raw_spin_lock(raw_spinlock_t *lock)
{
    debug_spin_lock_before(lock);
    arch_spin_lock(&lock->raw_lock);
    debug_spin_lock_after(lock);
}

#define arch_spin_is_locked(l)      queued_spin_is_locked(l)
#define arch_spin_is_contended(l)   queued_spin_is_contended(l)
#define arch_spin_value_unlocked(l) queued_spin_value_unlocked(l)
#define arch_spin_lock(l)           queued_spin_lock(l)
#define arch_spin_trylock(l)        queued_spin_trylock(l)
#define arch_spin_unlock(l)         queued_spin_unlock(l)
#define arch_spin_lock_flags(l, f)  queued_spin_lock(l)
#define arch_spin_unlock_wait(l)    queued_spin_unlock_wait(l)
```

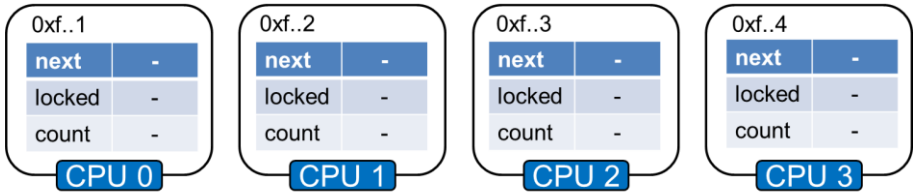
Queued-based Spinlocks

- spin_lock
  - fast path(first try optimization)
    - 바로 lock 잡을 수 있는 경우(lock 이 비어 있는 경우), mcs queueing 을 위한 복사 등 수행 없이 수행
    - cmpxchg 명령어를 사용하여 0 일 경우 (lock 사용 thread 없을 경우) locked 상태를 나타내는 0~7 bit 위치에 1 write 하여 lock 획득 하고 spinlock 종료

```
static __always_inline void queued_spin_lock(struct qspinlock *lock)
{
    u32 val;

    val = atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL);
    // qspinlock 의 val 이 0 인지 확인 하고 , 0 이라면 _Q_LOCKED_VAL 위치
    // 즉 0 bit 위치에 1 을 write 하며 wite 되기 전의 val 을 return 함
    // 0 이라면 .. pending 이든 뭐든 아무것도 없고 그냥 lock 안잡힌 상태
    if (likely(val == 0))
        return;
    queued_spin_lock_slowpath(lock, val);
}
```

```
typedef struct qspinlock {
        atomic_t        val;
} arch_spinlock_t;
```





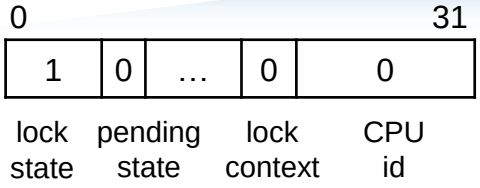
Queued-based Spinlocks

- spin_lock
  - fast path(first try optimization)
    - 바로 lock 잡을 수 있는 경우(lock 이 비어 있는 경우), mcs queueing 을 위한 복사 등 수행 없이 수행
    - cmpxchg 명령어를 사용하여 0 일 경우 (lock 사용 thread 없을 경우) locked 상태를 나타내는 0~7 bit 위치에 1 write 하여 lock 획득 하고 spinlock 종료
  - slowpath(pending bit optimization)
    - 아래 3 경우에 대하여 slowpath 수행
      - waiter 가 없을 경우, queue building 안함

```
static __always_inline void queued_spin_lock(struct qspinlock *lock)
{
    u32 val;

    val = atomic_cmpxchg_acquire(&lock->val, 0, _Q_LOCKED_VAL);
    // qspinlock 의 val 이 0 인지 확인 하고, 0 이라면 _Q_LOCKED_VAL 위치
    // 즉 0 bit 위치에 1 을 write 하며 wite 되기 전의 val 을 return 함
    // 0 이라면 .. pending 이든 뭐든 아무것도 없고 그냥 lock 안잡힌 상태
    if (likely(val == 0))
        return;
    queued_spin_lock_slowpath(lock, val);
}
```

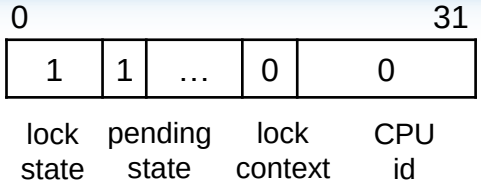
```
typedef struct qspinlock {
        atomic_t        val;
} arch_spinlock_t;
```

Case 1



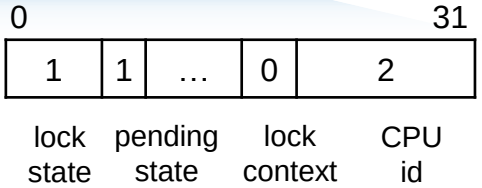
- lock 잡은 CPU 1 개(CPU 3), waiter 0 개, CPU 1 try

Case 2



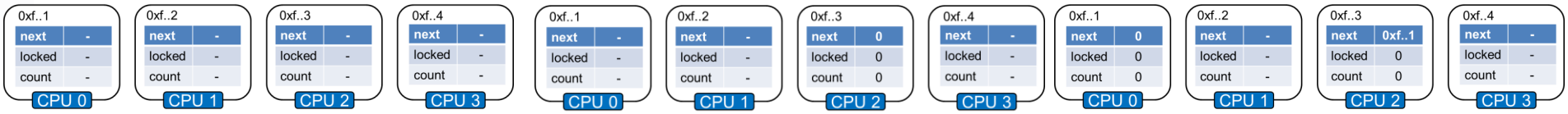
- lock 잡은 CPU 1 개(CPU3), waiter 1 개(CPU 1), CPU 2 try

Case 3



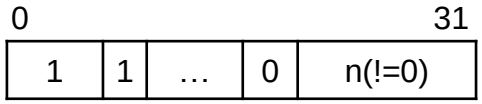
- lock 잡은 CPU 1 개(CPU3), waiter n 개(CPU1,CPU2), CPU 0 try

queue building 안함

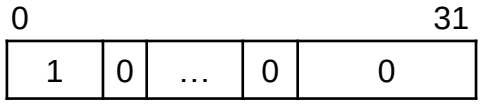


Queued-based Spinlocks

- spin_lock
  - slowpath(Case 1)
    - queue building 하기 전 case 1 (pending bit optimization) 상황인지 먼저 검사
      - _Q_LOCKED_VAL_, _Q_PENDING_VAL_ 이 모두 설정되어 있을 시 queue building 으로 바로 이동



      - _Q_LOCKED_VAL_ 만 설정 시, pending bit 추가

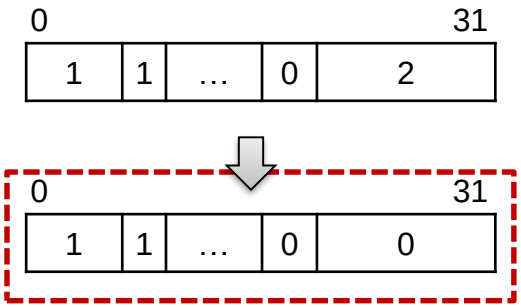


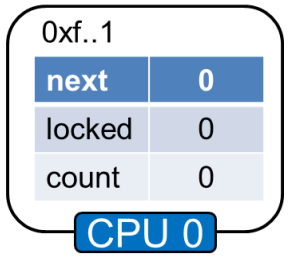
      - 계속 global lock 변수 확인하며 spinning

```
void queued_spin_lock_slowpath(struct qspinlock *lock, u32 val)
{
    …

    for (;;) {
        /*
         * If we observe any contention; queue.
         */
        if (val & ~_Q_LOCKED_MASK)
            goto queue;
        // val 이 locked 되고, pending state 인지 검사
        // 즉 현재 요청이 두번째 waiting CPU 라면 바로
        // queue building 해야 하므로 queue 로 이동
        // 아니라면 optimize 가능

        new = _Q_LOCKED_VAL;
        if (val == new)
            new |= _Q_PENDING_VAL;
        // pending optimization 수행
        // _Q_LOCKED_VAL 에 _Q_PENDING_VAL 추가하여 new 생성
        // 하나 thread 가 lock 을 잡은 상태에서 lock 요청이 왔다면
        // 즉, 첫번째 waiter 라면 queue building 하지 않고,
        // global lock 변수에 대해 spinning 하며 wait
        // (두개가 경쟁하는 상황이면 굳이 queue building 하여 나중에
        //  lock 풀릴 때, 1 cacheline miss 발생 추가할 필요가 없음)
        //
        //  for 문 spinning 하다가 lock 풀려서 pending bit clear 되고
        //  val 이 _Q_LOCKED_VAL 이 clear 된다면 lock 잡을수 있게됨

        /*
         * Acquire semantic is required here as the function may
         * return immediately if the lock was free.
         */
        old = atomic_cmpxchg_acquire(&lock->val, val, new);
        if (old == val)
            break;
        // lock->val 이 아직 val 과 같다면 즉 lock 된 상태라면 lock 변수에
        // pending state 설정된 new lock 을 write 후 break
        val = old;
    }

    /*
     * we won the trylock
     */
    if (new == _Q_LOCKED_VAL)
        return;
    // 위에 pending bit spinning 하다가 lock 잡은 상태임

    …

}
```

Queued-based Spinlocks

- spin_lock
  - slowpath(Case 2 & 3)
    - pending bit 가 설정되어 있을 경우, 2 개 이상의 lock waiter 가 이미 있는 경우 이므로 mcs queue building 시작
    - mcs_spinlock struct 형태의 per-CPU 변수 초기화
      - 어느 type mcs_spinlock 인지..
      - 다음 lock 변수 null 로 설정
    - 해당 TAIL 정보를 qspinlock 의 val 에 기록

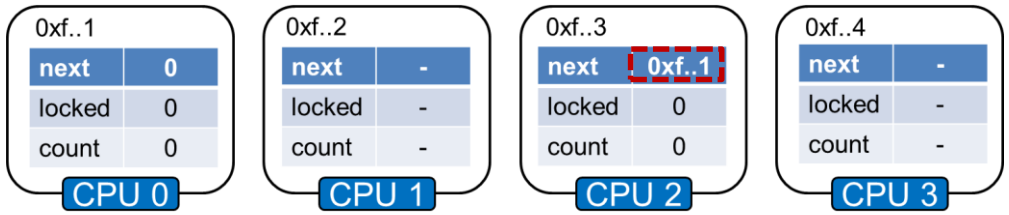





```
void queued_spin_lock_slowpath(struct qspinlock *lock, u32 val)
{
    …

    queue:
        // 세 번 째 thread 가 lock 을 잡 으 려 하 는 상 황 부 터 는 MCS queueing 으 로 동 작
        // Processor 별 로 생 성 되 있 는 mcs_spinlock[4] 중 첫 번 째 normal task context
        // 로 부 터 값 을 읽 어 들 임
        node = this_cpu_ptr(&mcs_nodes[0]);
        idx = node->count++;
        tail = encode_tail(smp_processor_id(), idx);
        // lock variable 의 16-32 bit 에 설 정 될 정 보 구 성

        node += idx;
        node->locked = 0;
        node->next = NULL;
        pv_init_node(node);

        /*
         * We touched a (possibly) cold cacheline in the per-cpu queue node;
         * attempt the trylock once more in the hope someone let go while we
         * weren't watching.
         */
        if (queued_spin_trylock(lock))
            goto release;
        // queue 에 추 가 전 lock 상 태 한 번 더 검 사
        /*
         * We have already touched the queueing cacheline; don't bother with
         * pending stuff.
         *
         * p,*,* -> n,*,*
         *
         * RELEASE, such that the stores to @node must be complete.
         */
        old = xchg_tail(lock, tail);
        // lock 변 수 에 계 산 한 tail 정 보 를 기 록
        next = NULL;

    …
}
```

Queued-based Spinlocks

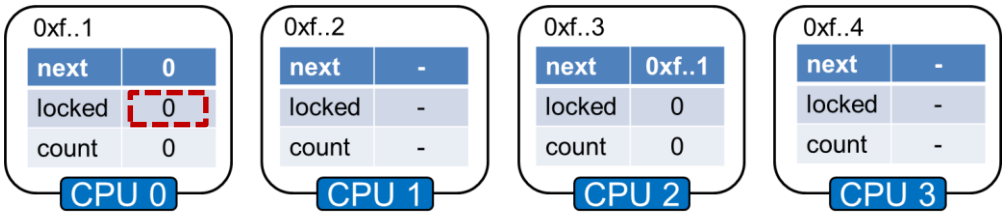
- spin_lock
  - slowpath(Case 2 & 3)
    - pending bit 가 설정되어 있을 경우, 2 개 이상의 lock waiter 가 이미 있는 경우 이므로 mcs queue building 시작
    - mcs_spinlock struct 형태의 per-CPU 변수 초기화
      - 어느 type mcs_spinlock 인지..
      - 다음 lock 변수 null 로 설정
    - 해당 TAIL 정보를 qspinlock 의 val 에 기록
    - 기존에 tail 정보가 있을 경우 next 에 현재 CPU 의 per-CPU 변수 주소 기록




```
void queued_spin_lock_slowpath(struct qspinlock *lock, u32 val)
{
    ...

    if (old & _Q_TAIL_MASK) {
        // queue 가 비어있는 상태가 아니라면 ...
        prev = decode_tail(old);
        /*
         * The above xchg_tail() is also a load of @lock which generates,
         * through decode_tail(), a pointer.
         *
         * The address dependency matches the RELEASE of xchg_tail()
         * such that the access to @prev must happen after.
         */
        smp_read_barrier_depends();

        WRITE_ONCE(prev->next, node);
        // 현재 processor 의 mcs_spinlock per-cpu variable 을 그전
        // variable 의 next 에 연결

        pv_wait_node(node, prev);
        arch_mcs_spin_lock_contended(&node->locked);
        // local 변수 보고 spinning 하며 대기
        /*
         * While waiting for the MCS lock, the next pointer may have
         * been set by another lock waiter. We optimistically load
         * the next pointer & prefetch the cacheline for writing
         * to reduce latency in the upcoming MCS unlock operation.
         */
        next = READ_ONCE(node->next);
        if (next)
            prefetchw(next);
        // 현재 mcs_spinlock per-cpu variable 의 next node 가 무언가 다른 processor
        // 의 per-cpu variable 에 연결된 상태라면 (다른 processor 도 접근 시도중이면 )
        // 그 processor 의 mcs_spinlock 을 미리 읽어옴
        // 추후 현재 processor 가 lock 잡고 일하고 unlock 하게 되면 next 에게
        // 너 차례라고 알려주어야 함
    }

    ...

}
```

Queued-based Spinlocks

- spin_lock
  - slowpath(Case 2 & 3)
    - pending bit 가 설정되어 있을 경우, 2 개 이상의 lock waiter 가 이미 있는 경우 이므로 mcs queue building 시작
    - mcs_spinlock struct 형태의 per-CPU 변수 초기화
      - 어느 type mcs_spinlock 인지..
      - 다음 lock 변수 null 로 설정
    - 해당 TAIL 정보를 qspinlock 의 val 에 기록
    - 기존에 tail 정보가 있을 경우 next 에 현재 CPU 의 per-CPU 변수 주소 기록
    - local per-CPU 변수에 대해 spinning 하며 lock 을 얻을 수 있는지 확인
      - lock 얻을 수 있게 되면 빠져나옴

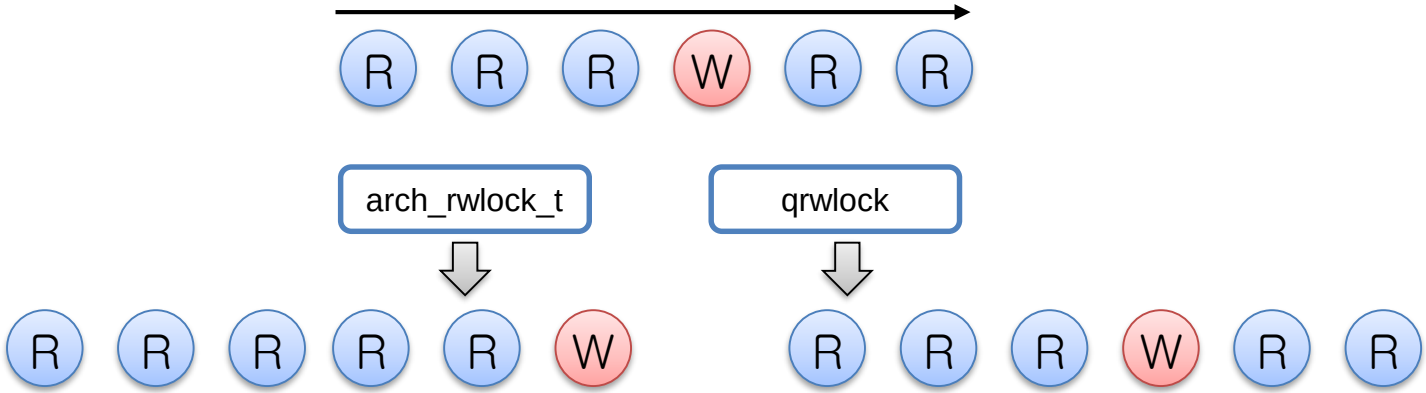



```
void queued_spin_lock_slowpath(struct qspinlock *lock, u32 val)
{
    ...

    if (old & _Q_TAIL_MASK) {
        // queue 가 비어있는 상태가 아니라면 ...
        prev = decode_tail(old);
        /*
         * The above xchg_tail() is also a load of @lock which generates,
         * through decode_tail(), a pointer.
         *
         * The address dependency matches the RELEASE of xchg_tail()
         * such that the access to @prev must happen after.
         */
        smp_read_barrier_depends();

        WRITE_ONCE(prev->next, node);
        // 현재 processor 의 mcs_spinlock per-cpu variable 를 그전
        // variable 의 next 에 연결

        pv_wait_node(node, prev);
        arch_mcs_spin_lock_contended(&node->locked);
        // local 변수 로고 spinning 하며 내가
        /*
        // 의 per-cpu variable 에 연결된 상태라면 (다른 processor 도 접근 시도중이면 )
        // 그 processor 의 mcs_spinlock 을 미리 읽어옴
        // 추후 현재 processor 가 lock 잡고 일하고 unlock 하게 되면 next 에게
        // 너차례라고 알려주어야 함
    }

    ...

}
```

```
#define arch_mcs_spin_lock_contended(l)
do {
    while (!(smp_load_acquire(l)))
        cpu_relax();
} while (0)
#endif
```

Reader-Wrier Spinlock

- reader/writer model 에서 read-mostly data 위한 lock 으로 writer-writer, writer-reader 사이에 상호 배제를 보장하며 reader-reader 간 임계 영역의 동시 접근 보장
  - 기존 rwlock 은 writer 보다 reader 에게 lock 을 우선적으로 주어 writer starvation 발생
  - x86 의 경우 queued-based rwlock 을 통해 writer watier 의 starvation 을 해결 수행 (v3.16)
    - writer waiter 가 있을 경우, lock 요청 queue 에 담긴 순서대로 writer 처리 후, reader 처리
    - interrupt context 의 reader 의 경우, lock queue 에 담지 않고 바로 처리



- 커널 내의 사용되는 곳들
  - jbd2 의 journal state 나타내는 journal_s 에 대한 접근 보호(j_state_lock)
  - ext4 의 extent 정보 나타내는 extent status tree 인 ext4_es_tree 에 대한 접근 보호(i_es_lock)
  - file owner 정보를 나타내는 fown_struct 에 대한 접근 보호(lock)
  - process list 를 탐색 할 때 쓰이는 tasklist_lock
  - …

Normal Reader-Wrier Spinlock

- 기존의 rwspin lock (patch 전)

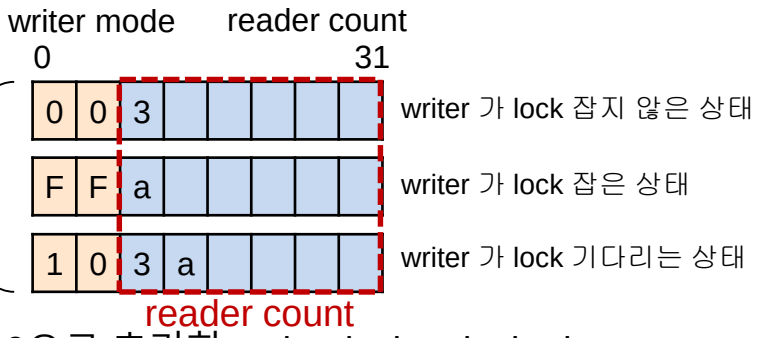
Queue-based rwlock

- data structures
  - qrwlock
    - cnts 를 통해 reader count ,writer lock 의 상태를 나타냄
      - 0-8 bit : writer 상태 (writer lock acquired, writer waiting)
      - 9-31 bit : critical section 에 진입해 있는 reader 의 수

```
typedef struct {
    arch_rwlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
#endif
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} rwlock_t;
```

```
typedef struct qrwlock {
    atomic_t        cnts;
    // reader counter, writer 여부
    // 0000 0000 0000 0000 0000 0000 0000 0000
    // <------->
    // writer 여부
    //           <--------------------------->
    //                      reder 개수
    arch_spinlock_t     wait_lock;
    // cnts 조작을 보호하기 위한 lock
} arch_rwlock_t;
```

```
struct __qrwlock {
    union {
        atomic_t cnts;
        struct {
#ifdef __LITTLE_ENDIAN
            u8 wmode;   /* Writer mode   */
            u8 rcnts[3];    /* Reader counts */
#else
            u8 rcnts[3];    /* Reader counts */
            u8 wmode;   /* Writer mode   */
#endif
        };
    };
    arch_spinlock_t lock;
};
```



- initialization
  - lockdep 관련 lock 의존성 필드 초기화 및 reader count 0으로 초기화, writer lock unlocked

```
# define rwlock_init(lock)                  \
    do { *(lock) = __RW_LOCK_UNLOCKED(lock); } while (0)

#define __RW_LOCK_UNLOCKED(lockname)                    \
    (rwlock_t)  {   .raw_lock = __ARCH_RW_LOCK_UNLOCKED,    \
                .magic = RWLOCK_MAGIC,          \
                .owner = SPINLOCK_OWNER_INIT,       \
                .owner_cpu = -1,            \
                RW_DEP_MAP_INIT(lockname) }
```

```
#define __ARCH_RW_LOCK_UNLOCKED {       \
    .cnts = ATOMIC_INIT(0),         \
    .wait_lock = __ARCH_SPIN_LOCK_UNLOCKED, \
}
```

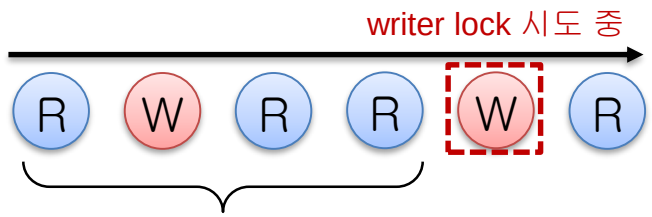
Queue-based rwlock

- write_lock
  - fast path
    - cnts 가 0이라면 lock 바로 잡음
      - writer 가 lock 을 잡고 있지 않고 reader count 가 0 이면 바로 lock 획득

```
static inline void queued_write_lock(struct qrwlock *lock)
{
    /* Optimize for the unfair lock case where the fair flag is 0. */
    if (atomic_cmpxchg_acquire(&lock->cnts, 0, _QW_LOCKED) == 0)
        return;
    // 0 일 경우 , writer index 도 비어있고 , reader 도 없으므로
    // reader ,writer 가 없을 경우 , writer 가 lock 바로 획득 가능
    queued_write_lock_slowpath(lock);
}
```

Queue-based rwlock

- write_lock
  - fast path
    - cnts 가 0이라면 lock 바로 잡음
      - writer 가 lock 을 잡고 있지 않고 reader count 가 0 이면 바로 lock 획득
  - slow path
    - lock 을 가진 writer 가 있을 경우, writer 의 수행이 끝날 때 까지 대기.
      - wmode 가 0 일 때까지 spinning
      - 선행 writer 수행이 끝나면 _QW_WAITING 설정
    - writer 가 끝난 후, 수행중인 reader 있을 경우, reader 가 모두 종료될 때까지 대기.
      - reader 가 남아있어 cts 값이 _QW_WAITING 이 아닐 경우 대기




```
void queued_write_lock_slowpath(struct qrwlock *lock)
{
    u32 cnts;

    /* Put the writer into the wait queue */
    arch_spin_lock(&lock->wait_lock);

    /* Try to acquire the lock directly if no reader is present */
    if (!atomic_read(&lock->cnts) &&
        (atomic_cmpxchg_acquire(&lock->cnts, 0, _QW_LOCKED) == 0))
        goto unlock;
        // writer 가 lock 얻을 수 있는지 한번 더 검사
    /*
     * Set the waiting flag to notify readers that a writer is pending,
     * or wait for a previous writer to go away.
     */
    for (;;) {
        struct __qrwlock *l = (struct __qrwlock *)lock;

        if (!READ_ONCE(l->wmode) &&
           (cmpxchg_relaxed(&l->wmode, 0, _QW_WAITING) == 0))
            break;
        // * writer 가 있을 경우,
        //   -> writer 수행이 끝날 때까지 spinning
        // * writer 가 없을 경우,
        //   -> writer 가 대기중임으로 나타내는 _QW_WAITING 설정 후, break

        cpu_relax();
    }

    /* When no more readers, set the locked flag */
    for (;;) {
        cnts = atomic_read(&lock->cnts);
        // reader 개수, writer flag 읽어옴
        if ((cnts == _QW_WAITING) &&
            (atomic_cmpxchg_acquire(&lock->cnts, _QW_WAITING,
                        _QW_LOCKED) == _QW_WAITING))
            break;
            // reader 가 다 나갈 때까지 기다리다가 lock 획득
            // (cnts 에 reader counter 가 0 이 되어 _QW_WAITING 만 남음)

        cpu_relax();
    }
unlock:
    arch_spin_unlock(&lock->wait_lock);
}
```

Queue-based rwlock

- read_lock
  - fast path
    - lock 을 잡고있는 writer 가 있거나, 대기중인 writer 가 있는 경우, 바로 reader lock 을 잡지 못함
      - reader counter 증가 후, _QW_MASK 를 통해 _QW_LOCKED, _QW_WAITING 여부 검사

```
static inline void queued_read_lock(struct qrwlock *lock)
{
    u32 cnts;

    cnts = atomic_add_return_acquire(_QR_BIAS, &lock->cnts);
    if (likely(!(cnts & _QW_WMASK)))
        return;
    // _QR_BIAS 만큼 증가 즉 reader counter 위치 1 증가
    // _QW_WMASK 즉 writer 가 없고 , _QW_WAITING 즉 writer waiter
    // 없다면 바로 lock 획득

    /* The slowpath will decrement the reader count, if necessary. */
    queued_read_lock_slowpath(lock, cnts);
}
```

Queue-based rwlock

- read_lock
  - fast path
    - lock 을 잡고있는 writer 가 있거나, 대기중인 writer 가 있는 경우, 바로 reader lock 을 잡지 못함
      - reader counter 증가 후, _QW_MASK 를 통해 _QW_LOCKED, _QW_WAITING 여부 검사
  - slow path
    - interrupt context 의 경우, writer 로부터 lock steal 가능
      - interrupt context 의 경우 lock 잡고 대기하지 않고 바로 spinning 하며 writer 가 lock 풀자마자 reader count update

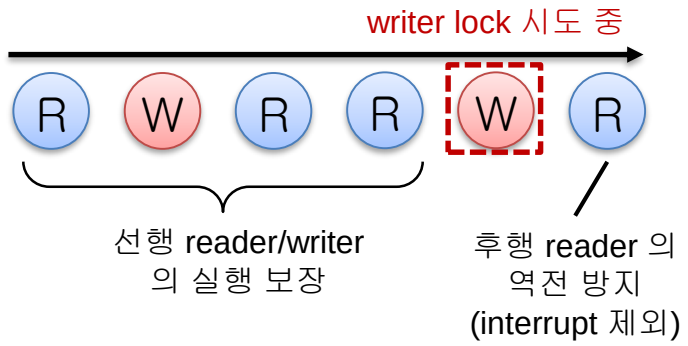
```
void queued_read_lock_slowpath(struct qrwlock *lock, u32 cnts)
{
    /*
     * Readers come here when they cannot get the lock without waiting
     */
    if (unlikely(in_interrupt())) {
        /*
         * Readers in interrupt context will get the lock immediately
         * if the writer is just waiting (not holding the lock yet).
         * The rspin_until_writer_unlock() function returns immediately
         * in this case. Otherwise, they will spin (with ACQUIRE
         * semantics) until the lock is available without waiting in
         * the queue.
         */
        rspin_until_writer_unlock(lock, cnts);
        // internal context 의 경우 , waiter 에 상관 없이 ,
        // writer lock 여부만 확인 후 , writer lock 끝나면 바로 진입
        // (wait_lock 잡지 않고 바로 writer 풀리길 대기 )
        return;
    }
```

```
static __always_inline void
rspin_until_writer_unlock(struct qrwlock *lock, u32 cnts)
{
    while ((cnts & _QW_WMASK) == _QW_LOCKED) {
        cpu_relax();
        cnts = atomic_read_acquire(&lock->cnts);
    }
}
```

```
     * that accesses can't leak upwards out of our subsequent critical
     * section in the case that the lock is currently held for write.
     */
    cnts = atomic_fetch_add_acquire(_QR_BIAS, &lock->cnts);
    rspin_until_writer_unlock(lock, cnts);
    // writer lock 이 풀리길 대기

    /*
     * Signal the next one in queue to become queue head
     */
    arch_spin_unlock(&lock->wait_lock);
}
```

Queue-based rwlock

- read_lock
  - fast path
    - lock 을 잡고있는 writer 가 있거나, 대기중인 writer 가 있는 경우, 바로 reader lock 을 잡지 못함
      - reader counter 증가 후, _QW_MASK 를 통해 _QW_LOCKED, _QW_WAITING 여부 검사
  - slow path
    - interrupt context 의 경우, writer 로부터 lock steal 가능
      - interrupt context 의 경우 lock 잡고 대기하지 않고 바로 spinning 하며 writer 가 lock 풀자마자 reader count update
    - interrupt context 가 아닌 경우, writer 와 같은 lock 을 잡으려 대기
      - 선행 writer 가 unlock 을 수행 해야 후행 reader 가 실행 가능




```
void queued_read_lock_slowpath(struct qrwlock *lock, u32 cnts)
{
    /*
     * Readers come here when they cannot get the lock without waiting
     */
    if (unlikely(in_interrupt())) {
        /*
         * Readers in interrupt context will get the lock immediately
         * if the writer is just waiting (not holding the lock yet).
         * The rspin_until_writer_unlock() function returns immediately
         * in this case. Otherwise, they will spin (with ACQUIRE
         * semantics) until the lock is available without waiting in
         * the queue.
         */
        rspin_until_writer_unlock(lock, cnts);
        // internal context 의 경우, waiter 에 상관 없이,
        // writer lock 여부만 확인 후, writer lock 끝나면 바로 진입
        // (wait_lock 잡지 않고 바로 writer 풀리길 대기)
        return;
    }
    atomic_sub(_QR_BIAS, &lock->cnts);
    // 일단 증가시켜 놓은것 뺌
    /*
     * Put the reader into the wait queue
     */
    arch_spin_lock(&lock->wait_lock);
    // interrupt context 가 아니므로 lock 잡고 순서 대기
    /*
     * The ACQUIRE semantics of the following spinning code ensure
     * that accesses can't leak upwards out of our subsequent critical
     * section in the case that the lock is currently held for write.
     */
    cnts = atomic_fetch_add_acquire(_QR_BIAS, &lock->cnts);
    rspin_until_writer_unlock(lock, cnts);
    // writer lock 이 풀리길 대기

    /*
     * Signal the next one in queue to become queue head
     */
    arch_spin_unlock(&lock->wait_lock);
}
```

Queue-based rwlock

- write_unlock
  - lock 변수의 write mode 위치에 0 write 즉 writer lock clear

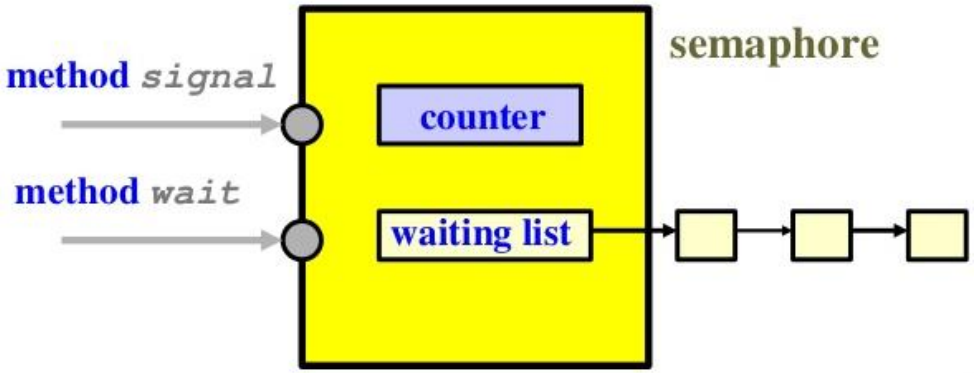
```
static inline void queued_write_unlock(struct qrwlock *lock)
{
    smp_store_release(__qrwlock_write_byte(lock), 0);
}
```

- read_unlock
  - lock 변수의 read count 위치의 reader 수 감소

```
static inline void queued_read_unlock(struct qrwlock *lock)
{
    /*
     * Atomically decrement the reader count
     */
    (void)atomic_sub_return_release(_QR_BIAS, &lock->cnts);
}
```

Semaphore

- critical section 에 동시에 여러 process 의 접근을 막기 위한 mechanism 으로 lock 을 잡지 못한 process 는 경우에 따라 sleep 된다.
- critical section 의 수행시간이 긴 경우에 적합하며, sleep 이 되면 안되는 interrupt context 에서 사용이 불가능하다.
- 커널 내의 사용되는 곳들
  - usb/gpu/video/mmc 등 주로 device driver 내에서 많이 사용



Semaphore

- data structure
  - semaphore 대기하는 task_struct 들을 wait_list 로 관리
  - critical section 진입 가능한 task 의 수를 count 로 나타냄
  - spinlock 을 통해 count & wait_list 관리
    - semaphore_waiter : task_structure 정보 & sleep 여부
    - semaphore 를 대기중인 wait_list 중 하나가 깨어남

```
struct semaphore {
    raw_spinlock_t      lock;
    // semaphore data 를 보호하기 위한 spinlock
    unsigned int        count;
    // critical section 에 들어갈 수 있는 process 의 수
    struct list_head    wait_list;
    // lock 얻기 위해 대기중인 process 들
};
```

```
struct semaphore_waiter {
    struct list_head list;
    struct task_struct *task;
    bool up;
    // 깨어났는지 여부
};
```

- initialization
  - static initialization
    - binary semaphore 용도 초기화

```
#define DEFINE_SEMAPHORE(name)  \
    struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)

#define __SEMAPHORE_INITIALIZER(name, n)                \
{                                       \
    .lock       = __RAW_SPIN_LOCK_UNLOCKED((name).lock),    \
    .count      = n,                                \
    .wait_list  = LIST_HEAD_INIT((name).wait_list),     \
}
```

  - dynamic initialization
    - counting semaphore 용도 초기화

```
static inline void sema_init(struct semaphore *sem, int val)
{
    static struct lock_class_key __key;
    *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
    lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
    // lock validator 관련 함수
}
```

Semaphore

- semaphore API
  - down
    - semaphore 를 잡기 위한 count 가 현재 가용 count 보다 큰지 검사.
      - 클 경우 semaphore 진입 가능
      - 작을 경우 sleep 및 wait_list 추가를 위해 __down 수행

```
void down(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    // semaphore 변수조정 전에 lock 잡고 IF 상태 저장
    if (likely(sem->count > 0))
        sem->count--;
    // semaphore 획득 가능하다면 획득하고 종료
    else
        __down(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
    // lock 풀고 IF 상태 복구
}

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

Semaphore

- semaphore API
  - down
    - semaphore 를 잡기 위한 count 가 현재 가용 count 보다 큰지 검사.
      - 클 경우 semaphore 진입 가능
      - 작을 경우 sleep 및 wait_list 추가를 위해 __down 수행
    - current task 를 wait_list 에 추가 및 sleep 상태로 설정(up ← false)
    - __down_common 함수로 sleep state 가 TASK_INTERRUPTABLE 또는 TASK_KILLABLE 이 아닐 경우는 괜찮지만 설정되어 있을 경우, 아직 처리되지 않은 signal 이 있으면 안되므로 종료
    - state 값에 따라 task 상태를 TASK_INTERRUPTABLE 또는 TASK_UNINTERRUPTABLE 로 설정 후
    - 지정된 시간(timeout)만큼 sleep
      - sleep 후 또는 signal 에 의해 깨어나면 up flag 가 설정되어 semaphore 잡을 수 있는지 검사
      - 아니면 다시 잠듬.

```
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);
    // semaphore 의 waiter list에 현재 list 추가
    waiter.task = current;
    // 현재 task 로 waiter 초기화
    waiter.up = false;
    // sleep 로 들어갈 것이므로 false
    // up 이 true 가 될 때 까지 아래 loop spining
    for (;;) {
        if (signal_pending_state(state, current))
            goto interrupted;
        // TASK_INTERRUPTIBLE 이 아니고, pending signal 이 없어야 함
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_current_state(state);
        // task 상태 설정
        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        // timeout 만큼 현재 task sleep 하도록 함
        // timeout 만큼 sleep 하다가 up 확인하고 signal/interrupt 확인하고
        // 다시 sleep
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }

 timed_out:
    list_del(&waiter.list);
    // semaphore 대기 list 에서 제거
    return -ETIME;

 interrupted:
    list_del(&waiter.list);
    return -EINTR;
}

static inline int signal_pending_state(long state, struct task_struct *p)
{
    if (!(state & (TASK_INTERRUPTIBLE | TASK_WAKEKILL)))
        return 0;
    // p 의 state 가 TASK_INTERRUPTIBLE 가 설정되어 있지 않아 interrupt 를
    // 받을 수 없거나 TASK_WAKEKILL 이 설정되어 있지 않아 fatal signal 이 와도
    // task 를 깨울 수 없다면 종료
    if (!signal_pending(p))
        return 0;
    // task 가 pending signal 이 없다면 종료
    // (받았지만 아직 처리 안된 signal 즉 받은 signal 이 있는지 검사)
    // 현재 받은 signal 이 없다면 TASK_INTERRUPTIBLE 로 sleep
    return (state & TASK_INTERRUPTIBLE) || __fatal_signal_pending(p);
    // TASK_INTERRUPTIBLE 상태이거나 SIGKILL signal 이 요청된 상태라면 TRUE
}
```

Semaphore

- semaphore API
  - up
    - semaphore 를 대기중인 task 가 없다면 단순 semaphore 증가
    - 있다면 list 의 첫번째 task_struct 를 깨워주어 semaphore 를 잡을 수 있도록 up flag 설정 (up ← true)

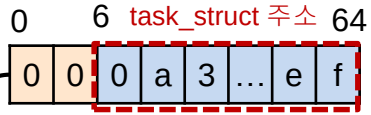
```
void up(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    // lock 잡고 semaphore 변수 조작, IF 저장
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;
    // 대기중인 task 가 없다면 count 증가
    else
        __up(sem);
    // 대기중인 task 있다면 깨움
    raw_spin_unlock_irqrestore(&sem->lock, flags);
    // lock 풀고 IF 복원
}

static noinline void __sched __up(struct semaphore *sem)
{
    struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
                        struct semaphore_waiter, list);
    // waiter list 들 중에서 첫번째 것 가져옴
    list_del(&waiter->list);
    // waiter list 에서 지우고
    waiter->up = true;
    // up 을 true 로 변경하여 깨어날 수 있도록 함
    wake_up_process(waiter->task);
    // sleep 하고 있을수도 있으므로
    // waiter task 를 TASK_RUNNING 상태로 바꾸고 run queue 의 맨 앞으로
}
```

Mutex

- critical section 에 동시에 여러 process 의 접근을 막기 위한 mechanism 으로 lock 을 잡지 못한 process 는 경우에 따라 sleep 되는 binary semaphore mechanism
  - 실제 구현은 semaphore 와 상당히 다름
  - semaphore 는 waiting list 의 task 로 reschedule 되어 결과적으로 context switch 발생하는 semaphore 와 달리 현재 mutex 는 최대한 이러한 context switch 막으려는 구조(mutex 에 OSQ lock 추가 v3.15 부터)
- 2016/9, 2017/1 patch 이후 구조 변경
  - 2016 년 5 월 patch 로 lock 상태를 나타내는 count 없어지는 등 구조 변경
- process 가 mutex 를 acquire 하려 할 때, 3 가지 path 가 있음
  - fast path
    - lock 을 잡을 수 있을 경우 그냥 count 0으로 변경하고, lock 잡고 풀 때 1 로 설정
  - mid path(MCS lock 기반 알고리즘의 optimistic spinning queue lock)
    - non-sleepable busy wait mutex
    - lock owner 가 이미 있을 경우… 나보다 higher priority 를 가진 ready to run process 가 없다면 MCS lock spinning 수행
    - waiting task 가 sleep 되지 않으므로 rescheduled 되지 않음 즉 expensive 한 context switch 없음
    - lock 잡고 있는 thread 가 돌고 있는 중이여야하며, lock 잡으려는 thread 가 선점 요청 안한 상태
  - slow path
    - semaphore 처럼 동작. lock 잡을 수 없으면 sleep, 나중에 reschedule
- 커널 내의 사용되는 곳들
  - kernel 내의 매우 다양한 곳에서 사용.
  - block device 관리 struct 인 block_device 에서 bd_mutex
  - file struct 에서 f_pos 접근을 관리하는 f_pos_lock
  - …

Mutex

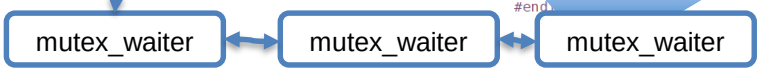
- data structure
  - mutex



MUTEX_FLAG_WAITERS : waiter 있음
MUTEX_FLAG_HANDOFF : handoff 과정
MUTEX_FLAG_PICKUP : handoff 과정

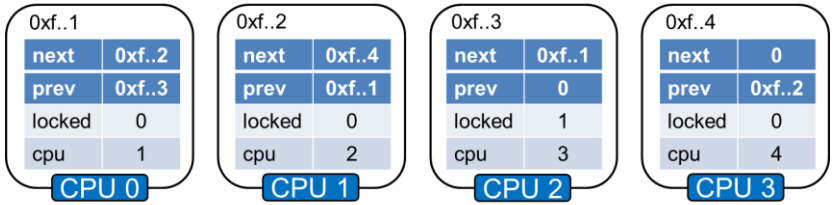
```
struct mutex {
    atomic_long_t      owner;
    // lock 을 잡은 task_struct 의 주소
    // owner 이 0 이면 즉 NULL 이면 lock 을 소유한
    // task_struct 가 없으므로 lock 이 free 상태
    //
    // 또한 address 가 L1_CACHE_BYTES align 되어 있으므로 (64 bit)
    // 0,1,2 bit 로 lock 의 상태정보를 나타냄
    //
    // 0 BIT 설정            - lock 잠김 mutex 를 얻으려 기다리는 동안 잠들어야 될때 설정
    // (MUTEX_FLAG_WAITERS) - lock 잡고있던 thread 는 lock release 시 이 bit 가 설정되어
    //                        있다면 waiter 를 wakeup 해주어야 함을 알수 있음
    //                      => 불필요한 연산 방지
    //                      => slowpath 들어간 thread 가 있을때 설정됨
    //
    // 1 BIT 설정            - 이미 lock 잡으려다 fail 후, 일정시간 sleep 했던 thread 가
    // (MUTEX_FLAG_HANDOFF)   깨어나 다시 lock 잡으려 시도할때 실패하게 되면
    //                        다시 또 sleep 들어가기 전에 이 bit 설정
    //                      - lock 잡고있던 thread 는 lock release 시 이 bit 가 설정되어
    //                        있다면 다른 새로 lock 을 잡으려는 thread 또는 Optimistic
    //                        spinning 중인 thread 가 lock 을 먼저 잡기 전지 않도록 하기
    //                        위해 단순히 owner 를 0 으로 clear 하는 것이 아니라, sleep
    //                        들어간 thread 를 깨우고, direct 로 wait_list 의 첫번째
    //                        thread 에게 소유권을 넘겨줌
    //                      => unfairness 방지
    //
    // 2 BIT 설정           - 1bit 설정 으로 lock 잡은 thread 가 lock 을 handoff
    // (MUTEX_FLAG_PICKUP)    하게될 때, 즉 건네 주게 될 때 1 BIT clear 하고,
    //                        pickup bit 를 새 owner 와 함께 설정함
    //                    => 같은 owner 가 같은 lock 에 또 접근시 오류 막기 위해
    //
    spinlock_t      wait_lock;
    // wait_list 에 대한 lock (slow path 용)
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
    // midpath 를 위한 MCS lock 알고리즘으로 동작하는 OSQ lock
#endif
    struct list_head    wait_list;
    // lock 잡기를 대기하는 process wait queue (mid path 용도)
#ifdef CONFIG_DEBUG_MUTEXES
    void            *magic;
    // mutex 관련 debugging 정보
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map  dep_map;
    // lock validator
#endif
};
```

```
struct mutex_waiter {
    struct list_head    list;
    // 다음 대기 mutex_waiter node 와 연결
    struct task_struct  *task;
    // 대기하는 thread 의 task_struct
    struct ww_acquire_ctx   *ww_ctx;
    // FIXME
    // wait-wound mutex 관련
#ifdef CONFIG_DEBUG_MUTEXES
    void            *magic;
    // mutex 관련 debug 정보
#endif
};
```



- optimistic_spin_node
  - qspinlock 과 같이 per-CPU variable 기반 동작(midpath)

```
static DEFINE_PER_CPU_SHARED_ALIGNED(struct optimistic_spin_node, osq_node);

struct optimistic_spin_node {
    struct optimistic_spin_node *next, *prev;
    // 다음 lock 대기자, 전 대기자
    int locked; /* 1 if lock acquired */
    // 0 - lock acquire 기다리는중 ..
    // 1 - lock acquire 성공
    int cpu; /* encoded CPU # + 1 value */
    // 현재 per-CPU optimistic_spin_node 가
    // 소속된 CPU 번호
};
```



- optimistic_spin_node
  - 현재 mutex 의 마지막 osq node 에 해당하는 processor id (실제 processor id + 1)
    e.g. tail ➔ 4

```
struct optimistic_spin_queue {
    /*
     * Stores an encoded value of the CPU # of the tail node in the queue.
     * If the queue is empty, then it's set to OSQ_UNLOCKED_VAL.
     */
    atomic_t tail;
    // lock 대기중인 per-CPU optimistic_spin_node list 에서
    // 마지막 node 가 속한 CPU 번호로 1 부터 시작하도록 설정됨
    // 0 은 OSQ_UNLOCKED_VAL 을 나타내는 값으로 osq queue 에 아무것도
    // 없을 경우 즉 처음으로 osq 에 진입한 경우임
};
```

Mutex

- initialization
  - spinlock, lockdep 관련 초기화 및 lock owner 를 0 으로 setting(task_struct 주소 0 으로)
  - static initialization

```
#define DEFINE_MUTEX(mutexname) \
    struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)


#define __MUTEX_INITIALIZER(lockname) \
        { .owner = ATOMIC_LONG_INIT(0) \
        , .wait_lock = __SPIN_LOCK_UNLOCKED(lockname.wait_lock) \
        , .wait_list = LIST_HEAD_INIT(lockname.wait_list) \
        __DEBUG_MUTEX_INITIALIZER(lockname) \
        __DEP_MAP_MUTEX_INITIALIZER(lockname) }
```

  - dynamic initialization

```
#define mutex_init(mutex)                       \
do {                                    \
    static struct lock_class_key __key;             \
                                        \
    __mutex_init((mutex), #mutex, &__key);              \
} while (0)


void
__mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key)
{
    atomic_long_set(&lock->owner, 0);
    spin_lock_init(&lock->wait_lock);
    INIT_LIST_HEAD(&lock->wait_list);
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    osq_lock_init(&lock->osq);
#endif

    debug_mutex_init(lock, name, key);
}

static inline void osq_lock_init(struct optimistic_spin_queue *lock)
{
    atomic_set(&lock->tail, OSQ_UNLOCKED_VAL);
}
```

Mutex

- mutex_lock
  - fast path
    - mutex lock 을 잡고있는 lock holder 가 현재 없는 경우 현재 task_struct 를 lock holder 로 바로 설정 가능
      - lock->owner 에 task_struct 설정 안됨
      - flag 들 설정 안됨

```
void __sched mutex_lock(struct mutex *lock)
{
    might_sleep();
    // debugging 목적 설정, 선점가능하도록 실행도중 scheduling 가능하도록 설정
    if (!__mutex_trylock_fast(lock)) // fast path 로 lock 을 얻으려 시도
        __mutex_lock_slowpath(lock);
    // 바로 lock 획득 불가능 한 경우, midpath/slowpath 로 ..
}
```

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

    if (!atomic_long_cmpxchg_acquire(&lock->owner, 0UL, curr))
        return true;
    // cmpxchg 연산을 수행, lock owner 가 0 과 같다면 즉 mutex 에 대한
    // 소유자가 없어서 lock 을 잡을 수 있는 상황이므로 currrent task_struct 의
    // 주소를 owner 에 설정 하고 빠르게 끝
    // fastpath 에서 lock 잡기 성공 !!

    return false;
}
```

Mutex

- mutex_lock
  - fast path
    - mutex lock 을 잡고있는 lock holder 가 현재 없는 경우 현재 task_struct 를 lock holder 로 바로 설정 가능
      - lock->owner 에 task_struct 설정 안됨
      - flag 들 설정 안됨
  - mid path / slow path
    - lock 을 바로 얻을 수 없는 경우, sleep 해야 되는 경우가 아니라면 MCS lock 기반으로 CPU 별 lock 변수 spinning
    - sleep 해야 되는 경우, mid path 를 포기하고 slow path 로 실행하여 semaphore 처럼 동작

```
void __sched mutex_lock(struct mutex *lock)
{
    might_sleep();
    // debugging 목적 설정, 선점가능하도록 실행도중 scheduling 가능하도록 설정
    if(!__mutex_trylock_fast(lock)) // fast path 로 lock 을 얻으려 시도
        __mutex_lock_slowpath(lock);
    // 바로 lock 획득 불가능 한 경우, midpath/slowpath 로 ..
}
```

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

    if (!atomic_long_cmpxchg_acquire(&lock->owner, 0UL, curr))
        return true;
    // cmpxchg 연산을 수행, lock owner 가 0 과 같다면 즉 mutex 에 대한
    // 소유자가 없어서 lock 을 잡을 수 있는 상황이므로 currrent task_struct 의
    // 주소를 owner 에 설정 하고 빠르게 끝
    // fastpath 에서 lock 잡기 성공 !!

    return false;
}
```

```
static noinline void __sched
__mutex_lock_slowpath(struct mutex *lock)
{
    __mutex_lock(lock, TASK_UNINTERRUPTIBLE, 0, NULL, _RET_IP_);
}


static int __sched
__mutex_lock(struct mutex *lock, long state, unsigned int subclass,
             struct lockdep_map *nest_lock, unsigned long ip)
{
    return __mutex_lock_common(lock, state, subclass, nest_lock, ip, NULL, false);
}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작

```
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)
{

    ...

    if (__mutex_trylock(lock) ||
        mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, NULL)) {
        // __mutex_trylock 으로 fastpath 한번 더 확인 후,
        //   - 성공이면 !NULL => true
        //   - 실패면 !owner 주소 => false => midpath 수행
        //
        // mutex_optimistic_spin 함수를 통해 mid path 수행
        //
        // => fastpath 또는 midpath 에서 lock 잡기 성공한다면 아래 코드 실행
        //
        /* got the lock, yay! */
        lock_acquired(&lock->dep_map, ip);
        // lock debugging 정보 기록
        if (use_ww_ctx && ww_ctx)
            ww_mutex_set_context_fastpath(ww, ww_ctx);
        // wait-wound mutex 관련
        // FIXME
        preempt_enable();
        return 0;
    }

    ...

}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작
    - mutex_can_spin_on_owner 를 통해 midpath 로 동작 가능한지 검사

```
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)
{

    ...

    if (!waiter) {
        /*
         * The purpose of the mutex_can_spin_on_owner() function is
         * to eliminate the overhead of osq_lock() and osq_unlock()
         * in case spinning isn't possible. As a waiter-spinner
         * is not going to take OSQ lock anyway, there is no need
         * to call mutex_can_spin_on_owner().
         */
        if (!mutex_can_spin_on_owner(lock))
            goto fail;
        // 아래 조건을 만족하는 상황만 midpath 가능
        //  - 현재 process 가 CPU 를 yield 하겠다는 선점요청을 한
        //    상태가 아니여야 함
        //  - 이미 lock 잡고있는 thread 가 다른 CPU 에서 돌고 있는 중이어야함
        //
        //  위에 해당되지 않는 놈을 미리 거름
        /*
         * In order to avoid a stampede of mutex spinners trying to
         * acquire the mutex all at once, the spinners need to take a
         * MCS (queued) lock first before spinning on the owner field.
         */
        if (!osq_lock(&lock->osq))
            goto fail;
        // mutex 의 osq lock 변수인 lock->osq 를 통해 midpath 수행
        //
        // midpath 에서 lock 잡은 경우 !true => 계속 실행
        // midpath 에서 slowpath 로 넘어가야 하는 경우 !false => fail 로 이동
    }

    ...

}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작
    - mutex_can_spin_on_owner 를 통해 midpath 로 동작 가능한지 검사
      - 현재 task 보다 우선순위가 높은 task 가 있어 CPU yield 해야 하는지 검사 schedule 되어야 한다면 mid path 실패
      - 현재 lock 을 잡고 있는 놈이 다른 CPU 에서 현재 동작중인 놈인지 검사 sleep 하고있다면 mid 실패

```
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)
{
```

```
static inline int mutex_can_spin_on_owner(struct mutex *lock)
{
    struct task_struct *owner;
    int retval = 1;

    if (need_resched())
        return 0;
    // 현재 task 가 CPU yield 요청 즉 preemption 요청이 설정되 있지 않아야
    // midpath 가 실행될 수 있음
    // 즉 현재 task 보다 우선순위가 높은 task 에 의해
    // CPU 요청이 있는 상태라면 midpath 포기하고, slowpath 로 전환
    rcu_read_lock();
    owner = __mutex_owner(lock);

    /*
     * As lock holder preemption issue, we both skip spinning if task is not
     * on cpu or its cpu is preempted
     */
    if (owner)
        retval = owner->on_cpu && !vcpu_is_preempted(task_cpu(owner));
    // 아래 조건을 만족해야
    //  - lock 이 잡혀 있는 경우, lock 잡고있는 놈이 돌고 있는 thread 가
    //    running 상태여야 함

    rcu_read_unlock();

    /*
     * 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;
}
```

```
    ...

}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작
    - mutex_can_spin_on_owner 를 통해 midpath 로 동작 가능한지 검사
      - 현재 task 보다 우선순위가 높은 task 가 있어 CPU yield 해야 하는지 검사 schedule 되어야 한다면 mid path 실패
      - 현재 lock 을 잡고 있는 놈이 다른 CPU 에서 현재 동작중인 놈인지 검사 sleep 하고있다면 mid 실패
    - mid path 검사 수행 후, mid path 진행이 가능한 상태라면 osq_lock 을 통해 MCS lock 동작

```
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)
{

    ...

    if (!waiter) {
        /*
         * The purpose of the mutex_can_spin_on_owner() function is
         * to eliminate the overhead of osq_lock() and osq_unlock()
         * in case spinning isn't possible. As a waiter-spinner
         * is not going to take OSQ lock anyway, there is no need
         * to call mutex_can_spin_on_owner().
         */
        if (!mutex_can_spin_on_owner(lock))
            goto fail;
        // 아래 조건을 만족하는 상황만 midpath 가능
        //  - 현재 process 가 CPU 를 yield 하겠다는 선점요청을 한
        //    상태가 아니여야 함
        //  - 이미 lock 잡고있는 thread 가 다른 CPU 에서 돌고 있는 중이어야함
        //
        //  위에 해당되지 않는 놈을 미리 거름
        /*
         * In order to avoid a stampede of mutex spinners trying to
         * acquire the mutex all at once, the spinners need to take a
         * MCS (queued) lock first before spinning on the owner field.
         */
        if (!osq_lock(&lock->osq))
            goto fail;
        // mutex 의 osq lock 변수인 lock->osq 를 통해 midpath 수행
        //
        // midpath 에서 lock 잡은 경우 !true => 계속 실행
        // midpath 에서 slowpath 로 넘어가야 하는 경우 !false => fail 로 이동
    }

    ...

}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작
    - mutex_can_spin_on_owner 를 통해 midpath 로 동작 가능한지 검사
      - 현재 task 보다 우선순위가 높은 task 가 있어 CPU yield 해야 하는지 검사 schedule 되어야 한다면 mid path 실패
      - 현재 lock 을 잡고 있는 놈이 다른 CPU 에서 현재 동작중인 놈인지 검사 sleep 하고있다면 mid 실패
    - mid path 검사 수행 후, mid path 진행이 가능한 상태라면 osq_lock 을 통해 MCS lock 동작
      - 현재 CPU 에 해당하는 per-CPU 변수 초기화 및 tail 변수에 현재 node 를 마지막으로 기록

```
bool osq_lock(struct optimistic_spin_queue *lock)
{
    struct optimistic_spin_node *node = this_cpu_ptr(&osq_node);
    // 현재 processor 의 optimistic_spin_node per-CPU 변수를 가져옴
    struct optimistic_spin_node *prev, *next;
    int curr = encode_cpu(smp_processor_id());
    // osq lock 의 cpu 번호 계산법에 따라 기존 cpu 번호 +1 값을 설정
    int old;

    node->locked = 0;
    // lock 을 잡으러 기다릴 것이므로 0 으로
    node->next = NULL;
    node->cpu = curr;
    // 현재 속한 CPU 번호 초기화

    /*
     * We need both ACQUIRE (pairs with corresponding RELEASE in
     * unlock() uncontended, or fastpath) and RELEASE (to publish
     * the node fields we just initialised) semantics when updating
     * the lock tail.
     */
    old = atomic_xchg(&lock->tail, curr);
    // osq wait queue 의 마지막에 추가될 것이므로 osq lock queue 의 tail 에
    // 현재 curr 값을 대입하고, 기존의 마지막 node 가 속한 CPU 번호를 알아온다.
    //
    // -- optimistic_spin_queue ----
    // |  tail          5->3     |
    // |             (old->curr) |
    // ---------------------------
    //
    //                                                  prev                                  node
    //           -- optimistic_spin_node -- <-| |------> -- optimistic_spin_node -- <-| |------> -- optimistic_spin_node --
    //           | prev           NULL |   --|--------| prev                |   --|--------| prev                |
    //           | next                |------        | next                |------        | next                |
    //           | locked         1    |              | locked         0    |              | locked         0    |
    //           | cpu            2    |              | cpu            5    |              | cpu          curr 6 |
    //           -------------------------            ------------------------            ------------------------
    // CPU0 ...  CPU1        CPU 2   ...    CPU3  CPU4     ...  CPU5              ...            CPU6

...

}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작
    - mutex_can_spin_on_owner 를 통해 midpath 로 동작 가능한지 검사
      - 현재 task 보다 우선순위가 높은 task 가 있어 CPU yield 해야 하는지 검사 schedule 되어야 한다면 mid path 실패
      - 현재 lock 을 잡고 있는 놈이 다른 CPU 에서 현재 동작중인 놈인지 검사 sleep 하고있다면 mid 실패
    - mid path 검사 수행 후, mid path 진행이 가능한 상태라면 osq_lock 을 통해 MCS lock 동작
      - 현재 CPU 에 해당하는 per-CPU 변수 초기화 및 tail 변수에 현재 node 를 마지막으로 기록
      - 그전 per-CPU osq node 와 연결 후 자신의 per-CPU 변수에 spinning 하며 lock 을 얻을 수 있는지 확인
      - 중간에 계속 midpath 불가능 조건 검사

```
bool osq_lock(struct optimistic_spin_queue *lock)
{
    ...
    prev = decode_cpu(old);
    node->prev = prev;
    WRITE_ONCE(prev->next, node);
    // prev 와 node 간 연결

    /*
     * Normally @prev is untouchable after the above store; because at that
     * moment unlock can proceed and wipe the node element from stack.
     *
     * However, since our nodes are static per-cpu storage, we're
     * guaranteed their existence -- this allows us to apply
     * cmpxchg in an attempt to undo our queueing.
     */

    while (!READ_ONCE(node->locked)) {
        // 이제 계속 자신의 local per-CPU variable 을 확인하며 spinning 하여
        // lock 을 얻을 수 있는지 확인
        /*
         * If we need to reschedule bail... so we can block.
         * Use vcpu_is_preempted() to avoid waiting for a preempted
         * lock holder:
         */
        if (need_resched() || vcpu_is_preempted(node_cpu(node->prev)))
            goto unqueue;
        // 현재 thread 가 선점되어야 하는경우 즉 현재 spinning 하는 thread
        // 보다 높은 우선순위의 task 가 요청을 하는 경우 midpath 를 포기하고
        // 빠져나가 slowpath 로 lock 대기
        cpu_relax();
        // NOP(No Operation) 수행
    }
    return true;
    // 위의 loop 를 빠져나온 것은 node->locked 가 1 로 변경 즉
    // lock acquire 성공한 것이므로 midpath 에서 lock 잡기 성공!!

    ...
}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작
    - mutex_can_spin_on_owner 를 통해 midpath 로 동작 가능한지 검사
      - 현재 task 보다 우선순위가 높은 task 가 있어 CPU yield 해야 하는지 검사 schedule 되어야 한다면 mid path 실패
      - 현재 lock 을 잡고 있는 놈이 다른 CPU 에서 현재 동작중인 놈인지 검사 sleep 하고있다면 mid 실패
    - mid path 검사 수행 후, mid path 진행이 가능한 상태라면 osq_lock 을 통해 MCS lock 동작
      - 현재 CPU 에 해당하는 per-CPU 변수 초기화 및 tail 변수에 현재 node 를 마지막으로 기록
      - 그전 per-CPU osq node 와 연결 후 자신의 per-CPU 변수에 spinning 하며 lock 을 얻을 수 있는지 확인
      - 중간에 계속 midpath 불가능 조건 검사
        - mid path 실패시, slow path 로이동하기 위해 queue 구성한 것 되돌림

```
bool osq_lock(struct optimistic_spin_queue *lock)
{
    ...
unqueue:

    //
    // midpath 에서 spinning 하며 lock 잡으려 하다가 현재 thread 보다
    // 우선순위 높은 놈에 의해 선점 요청이 들어온 경우 midpath 를 포기
    // 해야 하므로 queue 에서 삭제하고 slowpath 로 전환작업 수행
    for (;;) {
        if (prev->next == node &&
            cmpxchg(&prev->next, node, NULL) == node)
            break;
        // CPU2---CPU5---CPU6  =>  CPU2---CPU5

        // 위에 osq 재설정 과정이 성공하지 못한다면 아직 osq 에 node 를
        // 추가하려는 구성 중에 수행된 것일 수 있으므로
        if (smp_load_acquire(&node->locked))
            return true;

        cpu_relax();
        // NOP(No Operation) 수행
        prev = READ_ONCE(node->prev);
        // 순서 보장이 되었으니 다시 osq 재구성을 위해 prev 를 다시
        // 읽어들임
    }
    //
    // CPU 5 에서 선점 요청이 들어온 상태라면
    // -- optimistic_spin_queue ----
    // |   tail          5->3     |
    // |              (old->curr) |
    // ----------------------------
    //
    //                                                    node
    //             -- optimistic_spin_node -- <-| |------> -- optimistic_spin_node --
    //             |  prev             NULL |   --|--------|  prev                  |
    //             |  next                  |------         |  next                  |
    //             |  locked           1    |               |  locked          0     |
    //             |  cpu              2    |               |  cpu           curr 6  |
    //             ---------------------------               --------------------------
    // CPU0 ...  CPU1          CPU 2     ...  CPU3..CPU4..CPU5  ...     CPU6
    //

    next = osq_wait_next(lock, node, prev);
    // lock->tail 재설정 및 node->next를 가져옴
    //  e.g. CPU6
    if (!next)
        return false;
    // NULL 인 경우 내가 tail 이였으므로 더이상 작업 없이 종료
    // 즉 slowpath 로 넘어갈 준비 끝!!

    WRITE_ONCE(next->prev, prev);
    WRITE_ONCE(prev->next, next);
    // CPU 2 와 CPU6 서로 연결
    return false;
}
```

Mutex

- mutex_lock
  - fast path
  - mid path
    - mutex_optimistic_spin 함수를 통해 midpath 시작
    - mutex_can_spin_on_owner 를 통해 midpath 로 동작 가능한지 검사
      - 현재 task 보다 우선순위가 높은 task 가 있어 CPU yield 해야 하는지 검사 schedule 되어야 한다면 mid path 실패
      - 현재 lock 을 잡고 있는 놈이 다른 CPU 에서 현재 동작중인 놈인지 검사 sleep 하고있다면 mid 실패
    - mid path 검사 수행 후, mid path 진행이 가능한 상태라면 osq_lock 을 통해 MCS lock 동작
    - osq_lock 얻은 경우, lock owner 설정

```
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)
{

    ...

    // <midpath 성공한 경우!!>
    // case 1. waiter 가 NULL 이 아니거나 즉 wait-wound mutex 사용하는 경우
    //         FIXME
    // case 2. waiter 가 NULL 이며, osq 의 선두에 있는 놈이
    //         midpath 에서 lock 잡기 성공!!
    //         => 이제 owner 설정해 주어야 함
    for (;;) {
        struct task_struct *owner;

        /* Try to acquire the mutex... */
        owner = __mutex_trylock_or_owner(lock);
        if (!owner)
            break;
        // case 2.
        // osq 에서 spinning 하다가 자기 차례에서 lock 을 잡은 놈의
        // task_struct 를 owner 에 설정 해주고 break
        //
        // case 1.
        // wait-wound mutex 사용하는 경우
        // fastpath 한번 더 확인하고, midpath 과정 시작하기 위해 코드계속
        // FIXME
        //
        if (!mutex_spin_on_owner(lock, owner, ww_ctx, waiter))
            goto fail_unlock;
        cpu_relax();
    }

    if (!waiter)
        osq_unlock(&lock->osq);
        // case2.
        // osq lock 도 spinning 정상적으로 하다가 lock 잡고 끝났으므로
        // osq 에서 제거 및 spinning 하고 있는 다음 optimistic_spin_node
        // per-CPU 의 locked 에 1 을 write 하여 너차례라고 알림
    return true;
    // osq spinning 의 결과 성공적으로 osq lock 을 얻고,
    // mutex 의 owner 를 설정하였으니 true 로 함수 종료

fail_unlock:
    if (!waiter)
        osq_unlock(&lock->osq);

fail:
    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;
}
```

Mutex

- mutex_lock
  - fast path
  - mid path
  - slow path
    - semaphore 와 같은 동작 방식으로 현재 thread 를 sleep 시키고 schedule 함수를 통해 다른 task 수행.
    - 깨어나며 midpath 실행가능 검사 및 lock 획득 가능 검사

```
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)
{
    ...
    for (;;) {
        // 이제 sleep 했다가 ... 깨서 lock 획득 가능하나 확인했다가 ...
        // 다시 sleep 했다가 반복
        if (__mutex_trylock(lock))
            goto acquired;
        // mutex 얻을 수 있는지 확인

        if (unlikely(signal_pending_state(state, current))) {
            // mutex_lock 으로 호출시 , state 가 TASK_UNINTERRUPTIBLE 이므로
            // 관련 없지만 .
            // TASK_INTERRUPTIBLE 로 설정시 , mutex 를 기다리면 thread 가
            // signal 을 받아 mutex 획득을 중단 할 수 있음
            ret = -EINTR;
            goto err;
        }

        if (use_ww_ctx && ww_ctx && ww_ctx->acquired > 0) {
            // FIXME
            // wait-wound mutex 관련
            ret = __ww_mutex_lock_check_stamp(lock, &waiter, ww_ctx);
            if (ret)
                goto err;
        }
        // 이제 현재 thread 를 sleep 시키고 schedule 함수를 통해 다음
        // 수행해야 될 task 를 선택해야 하므로 wait_queue 를 지키고 있던
        // spinlock 을 풀음
        spin_unlock(&lock->wait_lock);
        schedule_preempt_disabled();
        // 다시 수행되게 되면 여기부터 수행됨
        if ((use_ww_ctx && ww_ctx) || !first) {
            // FIXME
            // wait-wound mutex 관련
            first = __mutex_waiter_is_first(lock, &waiter);
            if (first)
                __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
                // 현재 thread 가 wait_list 의 첫번째 놈이라면
                // MUTEX_FLAG_HANDOFF 설정하여 lock 잡고있던 놈이 lock 을
                // 바로전달 할 수 있게 함
        }

        set_current_state(state);
        // task 상태를 다시 설정
        if (__mutex_trylock(lock) ||
            (first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, &waiter)))
            break;
        // fastpath, midpath 로 lock acquire 시도
        spin_lock(&lock->wait_lock);
    }

    ...
}
```

Q & A

MOESI cache coherence protocol

- cache coherence protocol
  - multi processor system 에서 각 cache line 들이 동일한 데이터를 접근 할 수 있도록 보장해주는 mechanism
  - cache line 의 4가지 state가 존재
    - M(Modified) : cache 의 내용이 변경된 상태. 다른 processor 들의 해당 cache line Invalidate 화.
    - O(Owned) : 모든 processor가 최신 data를 가지지만, writeback되지는 않은 상태.
    - E(Exclusive) : 하나의 processor 만 해당 cache line을 가진 상태.
    - S(Shared) : 최소 하나의 processor 가 memory 에서 data를 읽어 들여 cache 에 저장한 상태.
    - I(Invalid) : 해당 processor 의 cache line 이 올바른 data 가 아닌 상태.