1.7 互斥量 mutex 的實現

1.7.1 mutex 的內核結構體

mutex 的定義及操作函數都在 Linux 內核檔 include\linux\mutex.h 中定義，如下：

```
// include/linux/mutex.h
struct mutex {
    /* 1: unlocked, 0: locked, negative: locked, possible waiters */
    atomic_t        count;          1.count要么等于1要么等于0
    spinlock_t      wait_lock;      2.mutex的实现要借助spinlock
    struct list_head    wait_list;3.等待mutex的进程放在这里
#if defined(CONFIG_DEBUG_MUTEXES) || defined(CONFIG_MUTEX_SPIN_ON_OWNER)
    struct task_struct  *owner;     4.owner只是用于调试或优化性能,
#endif                                并不是用来实现: 谁获得mutex就由谁释放mutex
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
#ifdef CONFIG_DEBUG_MUTEXES
    void            *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map  dep_map;
#endif
};
```

初始化 mutex 之後，就可以使用 mutex_lock 函數或其他衍生版本來獲取信號量，使用 mutex_unlock 函數釋放信號量。我們只分析 mutex_lock、mutex_unlock 函數的實現。

這裡要堪誤一下：前面的視頻裡我們說 mutex 中的 owner 是用來記錄獲得 mutex 的進程，以後必須由它來釋放 mutex。這是錯的！

從上面的代碼可知，owner 並不一定存在！

owner 有 2 個用途：debug(CONFIG_DEBUG_MUTEXES)或 spin_on_owner(CONFIG_MUTEX_SPIN_ON_OWNER)。

什麼叫 spin on owner？

我們使用 mutex 的目的一般是用來保護一小段代碼，這段代碼運行的時間很快。這意味著一個獲得 mutex 的進程，可能很快就會釋放掉 mutex。

針對這點可以進行優化，特別是當前獲得 mutex 的進程是在別的 CPU 上運行、並且“我”是唯一等待這個 mutex 的進程。在這種情況下，那“我”就原地 spin 等待吧：懶得去休眠了，休眠又喚醒就太慢了。

所以，mutex 是做了特殊的優化，比 semaphore 效率更高。但是在代碼上，並沒有要求“誰獲得 mutex，就必須由誰釋放 mutex”，只是在使用慣例上是“誰獲得 mutex，就必須由誰釋放 mutex”。

1.7.2 mutex_lock 函數的實現

1.7.2.1 fastpath

mutex 的設計非常精巧，比 semaphore 複雜，但是更高效。

首先要知道 mutex 的操作函數中有 fastpath、slowpath 兩條路徑(快速、慢速)：如果 fastpath 成功，就不必使用 slowpath。

怎麼理解？

這需要把 metex 中的 count 值再擴展一下，之前說它只有 1、0 兩個取值，1 表示 unlocked，0 表示 locked，還有一類值“負數”表示“locked，並且可能有其他程式在等待”。

代碼如下：

```
// kernel/locking/mutex.c
void __sched mutex_lock(struct mutex *lock)
{
    might_sleep();
    /*
     * The locking fastpath is the 1->0 transition from
     * 'unlocked' into 'locked' state.
     */ 1.先尝试fastpath
    __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath); 2.如果失败再用slowpath
    mutex_set_owner(lock); 3.执行到这肯定是成功了， 设置owner
}
```

先看看 fastpath 的函數：__mutex_fastpath_lock，這個函數在下面 2 個檔中都有定義：

```
include/asm-generic/mutex-xchg.h
include/asm-generic/mutex-dec.h
```

使用哪一個檔呢？看看 arch/arm/include/asm/mutex.h，內容如下：

```
#if __LINUX_ARM_ARCH__ < 6
#include <asm-generic/mutex-xchg.h>
#else
#include <asm-generic/mutex-dec.h>
#endif
```

所以，對於 ARMv6 以下的架構，使用 include/asm-generic/mutex-xchg.h 中的__mutex_fastpath_lock 函數；對於 ARMv6 及以上的架構，使用 include/asm-generic/mutex-dec.h 中的__mutex_fastpath_lock 函數。這 2 個檔中的__mutex_fastpath_lock 函數是類似的，mutex-dec.h 中的代碼如下：

```
// include/asm-generic/mutex-dec.h
static inline void
__mutex_fastpath_lock(atomic_t *count, void (*fail_fn)(atomic_t *))
{
    if (unlikely(atomic_dec_return_acquire(count) < 0)) 如果count初始值为1,
        fail_fn(count); 减1后为0, if条件不成立,成功获得mutex
}                       减1后<0,则使用slowpath休眠等待mutex
                        就是__mutex_lock_slowpath
```

大部分情況下，mutex 當前值都是 1，所以通過 fastpath 函數可以非常快速地獲得 mutex。

1.7.2.2 slowpath

如果 mutex 當前值是 0 或負數，則需要調用__mutex_lock_slowpath 慢慢處理：可能會休眠等待。

```
// kernel/locking/mutex.c
__visible void __sched
__mutex_lock_slowpath(atomic_t *lock_count)
{
    struct mutex *lock = container_of(lock_count, struct mutex, count);

    __mutex_lock_common(lock, TASK_UNINTERRUPTIBLE, 0,
                        NULL, _RET_IP_, NULL, 0);
}
```

核心函数

__mutex_lock_common 函數也是在內核檔 kernel/locking/mutex.c 中實現的，下麵分段講解。

① 分析第一段代碼：

```
504: static __always_inline int __sched
505: __mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
506:             struct lockdep_map *nest_lock, unsigned long ip,
507:             struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
508: {
509:     struct task_struct *task = current;
510:     struct mutex_waiter waiter;
511:     unsigned long flags;
512:     int ret;
513:
514:     if (use_ww_ctx) {
515:         struct ww_mutex *ww = container_of(lock, struct ww_mutex, base);
516:         if (unlikely(ww_ctx == READ_ONCE(ww->ctx)))
517:             return -EALREADY;
518:     }
519:
520:     preempt_disable();
521:     mutex_acquire_nest(&lock->dep_map, subclass, 0, nest_lock, ip);
522:
523:     if (mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx)) {
524:         /* got the lock, yay! */
525:         preempt_enable();
526:         return 0;
527:     }
528:
```

use_ww_ctx等于0，不走这分支

先禁止preempt

optimistic乐观的，

也许其他CPU获得了mutex，

它会很快释放，可能不需要休眠

这个函数不是重点，

这是一个优化而已

② 分析第二段代碼：

```
528:
529:     spin_lock_mutex(&lock->wait_lock, flags);
530:
531:     /*
532:      * Once more, try to acquire the lock. Only try-lock the mutex if
533:      * it is unlocked to reduce unnecessary xchg() operations.
534:      */
535:     if (!mutex_is_locked(lock) &&
536:         (atomic_xchg_acquire(&lock->count, 0) == 1))
537:         goto ↓skip_wait;
538:
```

获得spinlock

如果count==1，

那就不需要等待了

③ 分析第三段代碼：

```
539:     debug_mutex_lock_common(lock, &waiter);
540:     debug_mutex_add_waiter(lock, &waiter, task);
541:
542:     /* add waiting tasks to the end of the waitqueue (FIFO): */
543:     list_add_tail(&waiter.list, &lock->wait_list);
544:     waiter.task = task;
545:
546:     lock_contended(&lock->dep_map, ip);
547:
```

把当前进程放入mutex的wait_list

這個 wait_list 是 FIFO(Firt In Firs Out)，誰先排隊，誰就可以先得到 mutex。

④ 分析第四段代碼：for 迴圈，這是重點

```
548:     for (;;) {
549:         /*
550:          * Lets try to take the lock again - this is needed even if
551:          * we get here for the first time (shortly after failing to
552:          * acquire the lock), to make sure that we get a wakeup once
553:          * it's unlocked. Later on, if we sleep, this is the
554:          * operation that gives us the lock. We xchg it to -1, so
555:          * that when we release the lock, we properly wake up the
556:          * other waiters. We only attempt the xchg if the count is
557:          * non-negative in order to avoid unnecessary xchg operations:
558:          */
559:         if (atomic_read(&lock->count) >= 0 &&
560:             (atomic_xchg_acquire(&lock->count, -1) == 1))
561:             break;
562:
563:         /*
564:          * got a signal? (This code gets eliminated in the
565:          * TASK_UNINTERRUPTIBLE case.)
566:          */
567:         if (unlikely(signal_pending_state(state, task))) {
568:             ret = -EINTR;
569:             goto ↓err;
570:         }
571:
572:         if (use_ww_ctx && ww_ctx->acquired > 0) {
573:             ret = __ww_mutex_lock_check_stamp(lock, ww_ctx);
574:             if (ret)
575:                 goto ↓err;
576:         }
577:
578:         __set_task_state(task, state);
579:
580:         /* didn't get the lock, go to sleep: */
581:         spin_unlock_mutex(&lock->wait_lock, flags);
582:         schedule_preempt_disabled();
583:         spin_lock_mutex(&lock->wait_lock, flags);
584:     } « end for ;; »
```

如果count==1, 把它设为-1(被我lock了),break退出循环(成功了);
如果count==0,把它设为-1(被别人lock了),继续往下走;
如果count<0,它表示本来就被别人lock了,不用再设为-1,继续往下走

有信号? 那就退出吧

把当前进程设为非RUNNING状态

释放spinlock

主动发起调度

再次获得spinlock

能运行到这里, 是被信号或者mutex_unlock唤醒的, 在for循环里的再次判断

⑤ 分析第五段代碼：收尾工作

```
585:     __set_task_state(task, TASK_RUNNING);   能执行到这里,表示获得了mutex
586:                                              设置当前进程为RUNNING
587:     mutex_remove_waiter(lock, &waiter, task);
588:     /* set it to 0 if there are no waiters left: */
589:     if (likely(list_empty(&lock->wait_list)))
590:         atomic_set(&lock->count, 0);
591:     debug_mutex_free_waiter(&waiter);
592:
593: skip_wait:
594:     /* got the lock - cleanup and rejoice! */
595:     lock_acquired(&lock->dep_map, ip);
596:     mutex_set_owner(lock);
597:
598:     if (use_ww_ctx) {
599:         struct ww_mutex *ww = container_of(lock, struct ww_mutex, base);
600:         ww_mutex_set_context_slowpath(ww, ww_ctx);
601:     }
602:
603:     spin_unlock_mutex(&lock->wait_lock, flags);
604:     preempt_enable();
605:     return 0;
606:
```

从mutex的wait_list中删除本进程

如果wait_list为空,表示无人等待mutex,把count设为0

设置owner为当前进程

use_ww_ctx为0,不走这分支

释放spinlock

使能preempt

1.7.3 mutex_unlock 函數的實現

mutex_unlock 函數中也有 fastpath、slowpath 兩條路徑(快速、慢速)：如果 fastpath 成功，就不必使用 slowpath。

代碼如下：

```
// kernel/locking/mutex.c
void __sched mutex_unlock(struct mutex *lock)
{
    /*
     * The unlocking fastpath is the 0->1 transition from 'locked'
     * into 'unlocked' state:
     */
#ifndef CONFIG_DEBUG_MUTEXES
    /*
     * When debugging is enabled we must not clear the owner before time,
     * the slow path will always be taken, and that clears the owner field
     * after verifying that it was indeed current.
     */
    mutex_clear_owner(lock);
#endif
    __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
}
```

先使用fashpath

如果失败再使用slowpath

1.7.3.1 fastpath

先看看 fastpath 的函數：__mutex_fastpath_lock，這個函數在下面 2 個檔中都有定義：

```
include/asm-generic/mutex-xchg.h
include/asm-generic/mutex-dec.h
```

使用哪一個檔呢？看看 arch/arm/include/asm/mutex.h，內容如下：

```
#if __LINUX_ARM_ARCH__ < 6
#include <asm-generic/mutex-xchg.h>
#else
#include <asm-generic/mutex-dec.h>
#endif
```

所以，對於 ARMv6 以下的架構，使用 include/asm-generic/mutex-xchg.h 中的__mutex_fastpath_lock 函數；對於 ARMv6 及以上的架構，使用 include/asm-generic/mutex-dec.h 中的__mutex_fastpath_lock 函數。這 2 個檔中的__mutex_fastpath_lock 函數是類似的，mutex-dec.h 中的代碼如下：

```
// include/asm-generic/mutex-dec.h
static inline void
__mutex_fastpath_unlock(atomic_t *count, void (*fail_fn)(atomic_t *))
{
    if (unlikely(atomic_inc_return_release(count) <= 0))
        fail_fn(count);
}
```

加1后，
新值等于1，表示无人等待mutex,直接返回，
新值<=0,需要调用slowpath唤醒别的进程

就是__mutex_unlock_slowpath

大部分情況下，加 1 後 mutex 的值都是 1，表示無人等待 mutex，所以通過 fastpath 函數直接增加 mutex 的 count 值為 1 就可以了。

如果 mutex 的值加 1 後還是小於等於 0，就表示有人在等待 mutex，需要去 wait_list 把它取出喚醒，這需要用到 slowpath 的函數：__mutex_unlock_slowpath。

1.7.3.2 slowpath

如果 mutex 當前值是 0 或負數，則需要調用__mutex_unlock_slowpath 慢慢處理：需要喚醒其他進程。

```
// kernel/locking/mutex.c
/*
 * Release the lock, slowpath:
 */
__visible void
__mutex_unlock_slowpath(atomic_t *lock_count)
{
    struct mutex *lock = container_of(lock_count, struct mutex, count);

    __mutex_unlock_common_slowpath(lock, 1);
}
```

核心函数

__mutex_unlock_common_slowpath 函數代碼如下，主要工作就是從 wait_list 中取出並喚醒第 1 個進程：

```
// kernel/locking/mutex.c
static inline void
__mutex_unlock_common_slowpath(struct mutex *lock, int nested)
{
    unsigned long flags;
    WAKE_Q(wake_q);

    /*
     * As a performance measurement, release the lock before doing other
     * wakeup related duties to follow. This allows other tasks to acquire
     * the lock sooner, while still handling cleanups in past unlock calls.
     * This can be done as we do not enforce strict equivalence between the
     * mutex counter and wait_list.
     *
     *
     * Some architectures leave the lock unlocked in the fastpath failure
     * case, others need to leave it locked. In the later case we have to
     * unlock it here - as the lock counter is currently 0 or negative.
     */
    if (__mutex_slowpath_needs_to_unlock())
        atomic_set(&lock->count, 1);

    spin_lock_mutex(&lock->wait_lock, flags);
    mutex_release(&lock->dep_map, nested, _RET_IP_);
    debug_mutex_unlock(lock);

    if (!list_empty(&lock->wait_list)) {
        /* get the first entry from the wait-list: */
        struct mutex_waiter *waiter =
                list_entry(lock->wait_list.next,
                       struct mutex_waiter, list);

        debug_mutex_wake_waiter(lock, waiter);
        wake_q_add(&wake_q, waiter->task);
    }

    spin_unlock_mutex(&lock->wait_lock, flags);
    wake_up_q(&wake_q);
} « end __mutex_unlock_common_slowpath »
```

执行到这里count被设为1了

获取spinlock

从wait_list中
取出第一个进程

释放pinlock

唤醒第一个等待的进程