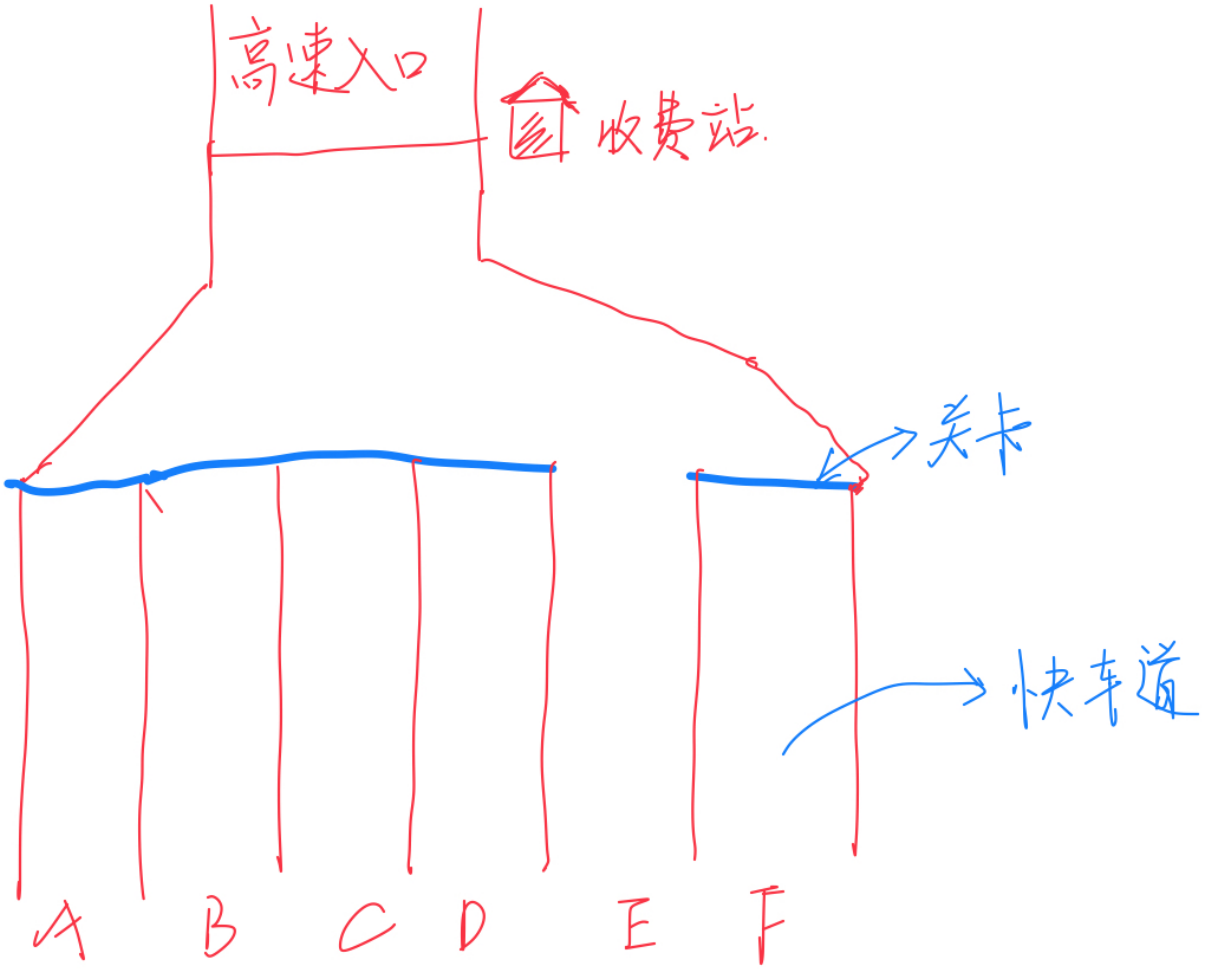
Mutex互斥量机制 Linux源码解读

Linux 4-10.1

软件工程学院 储震坤 20318068

一、实例分析

假设在现实生活中有下面的场景：
如图所示，在高速路入口处，有多条快车道在收费站并入高速公路，每条快车道上有多辆车在排队等侯在每条快车道的尽头又有一道关卡。每条快车道上的关卡和高速公路入口的收费站都是 **一次只允许一辆车经过** 。要怎么安排才能使保证对于整体来说高效又相对公平呢?



现在规定：所有快车道尽头的关卡， **每次只能有一个打开** ，其他关卡都保持关闭。如果在某条快车道的尽头关卡已经打开，但是第一辆车因为某些原因没有进入收费站而是选择进入服务区，那么等到这辆车从服务区休息完出来的时候， **可以直接从服务区进入高速公路不用重新排队** 。

在这种规定的情况下，就能很好的实现各条快车道上的车有序进入高速公路。对于那些自身所在快车道尽头关卡关闭的车辆，就只用关注自己所在车道上的关卡有没有开；而自身所在车道尽头的关卡打开的车辆，他们可以在关卡面前等待收费站的车辆缴费结束，一旦收费站的车辆缴费结束进入高速公路，这些车辆可以直接开进收费站。
理解了上面的实例，有助于我们思考 Mutex互斥量的机制。

二、Mutext运行机制

Linux中Mutex锁的特点：

> ### Mutex使用规则
>
> - mutex锁只能通过接口使用
> - 在同一时刻内只能有一个任务（使用task表示）持有该锁
> - **只有持有锁的进程才能解锁**
> - 不允许重复加锁和递归加锁
> - mutex不能在中断或者其他不允许睡眠的地方使用
> - **获取锁的进程不能退出，这会导致其他请求锁的进程饿死**

mutex锁和其他锁一样，也存在死锁的情况。在Linux中为了解决mutex的死锁问题提出了 ww_mutex锁，即 wound_wait 伤害等待机制，进程1、2同时申请锁A和锁B时，进程1拿到了锁A，进程2拿到了锁B，这时根据伤害等待机制，相对较年轻的进程（假设是进程B）会主动让出已经拿到的锁，等待对方执行完离开临界区之后再申请拿到全部的锁。这里先不对 ww_mutex 做出讨论，只研究mutex锁的机制。

接下来从一个申请使用Mutex锁的进程视角来解读mutex锁的机制

仍然是开头的快车道和高速公路的例子，现在你是一个在快车道上需要进入高速公路的司机，进入高速公路的流程是什么？

首先，你可以看看是不是肉眼能及的地方都只有你自己的一辆车，如果是，那么恭喜你，你可以直接将车开上高速公路；如果不是，你得先看看你所在的这条快车道的关卡有没有开，如果开了的话你就可以不用换挡只用降点油门再带一下刹车，因为你知道前面收费站的汽车很快就能办完缴费手续，你只需要稍微等待一下马上就是你了，如果你面前的关卡没有打开而是关闭的，或者说你前面还有很多辆车，那么你就要停下来挂空挡等待了。此时情况还可以接受，如果情况实在是太糟糕的话，你还可以选择进入服务区，睡一觉，等接到路况转好的通知再上路排队。

其实这一套流程和进程申请mutex锁是一样的，当一个进程申请mutex锁时：

1. **程序会先检查mutex锁是不是有其他进程申请、有没有其他进程在使用？尝试直接获取锁**
   如果都没有的话就会直接把锁给申请的进程，这就对应前面司机旁边没有其他任何一辆车的情况，司机可以直接开上高速公路，进程也可以直接获得mutex锁。
2. **如果不能直接获取锁的话，进程申请获得 osq锁，采取循环忙等方式等待osq锁**
   这里可以把osq锁想象成快车道尽头的关卡，申请mutex锁的进程先被挡在osq锁，想上高速公路的司机先被挡在快车道尽头的关卡。司机想要到高速公路收费站就必须先等自己这条快车道的关卡打开，进程想要获得mutex锁就必须先获得osq锁。司机在关卡前等待前面的汽车全部通过并且关卡打开的时候发动机并没有关闭，申请osq锁的进程也一直在循环忙等。
3. **获取osq锁之后，进程循环等待mutex锁的持有者释放锁并将锁拿过来**
   当进程获得osq锁的时候，就进入 **乐观自旋** ，等待mutex锁的持有者放锁，一旦放锁就直接把锁拿过来。就像在打开的关卡面前等待前面收费站的汽车离开之后直接进入收费站一样。但是如果前面的汽车开走之后在服务区收到新的路况通知的司机又立马开车上来，这位司机的车子可以在我们前面进入收费站，我们需要等待从服务区进来的汽车进入高速公路后才能进入收费站，这也就对应了如果有因为正在乐观自旋的进程因为内核调度导致睡眠，后面又被唤醒的时候，即使没有获得osq锁也可以直接从前一个mutex锁持有者哪里拿到锁。
   需要特别注意的是，即使进程在乐观自旋也是可以被调度的，只是在获得乐观自旋的申请过程加了锁不能被调度，获得锁之后还是可以被调度的。
   对于正在乐观自旋的进程还有一个情况需要注意，如果mutex锁的持有者对调度或者睡眠了，那么正在乐观自旋的进程，短时间内拿不到锁，会进入wait_list队列并睡眠。

对于mutex锁的释放过程

锁持有者在放锁的时候先尝试快速放锁，快速放锁不成功在进入慢速放锁
慢速放锁的流程可以想象成收费站的管理员，当正在收费站里面的汽车交完费准备离开的时候，管理员会查看是不是有之前已经到了关卡前面但是现在在服务区休息的司机，如果有的话，管理员会先通知在服务区休息的司机让司机直接进入收费站。如果没有的话，管理员会让在已经打开的关卡门口等待的汽车进入收费站。对比上面过程，mutex锁的慢速释放，就是先查看是否有进程因为自旋失败进入睡眠，如果有先将锁转交给这个进程再唤醒它；如果没有，将锁的相关信息清零完成释放。

三、Mutex的原理与实现

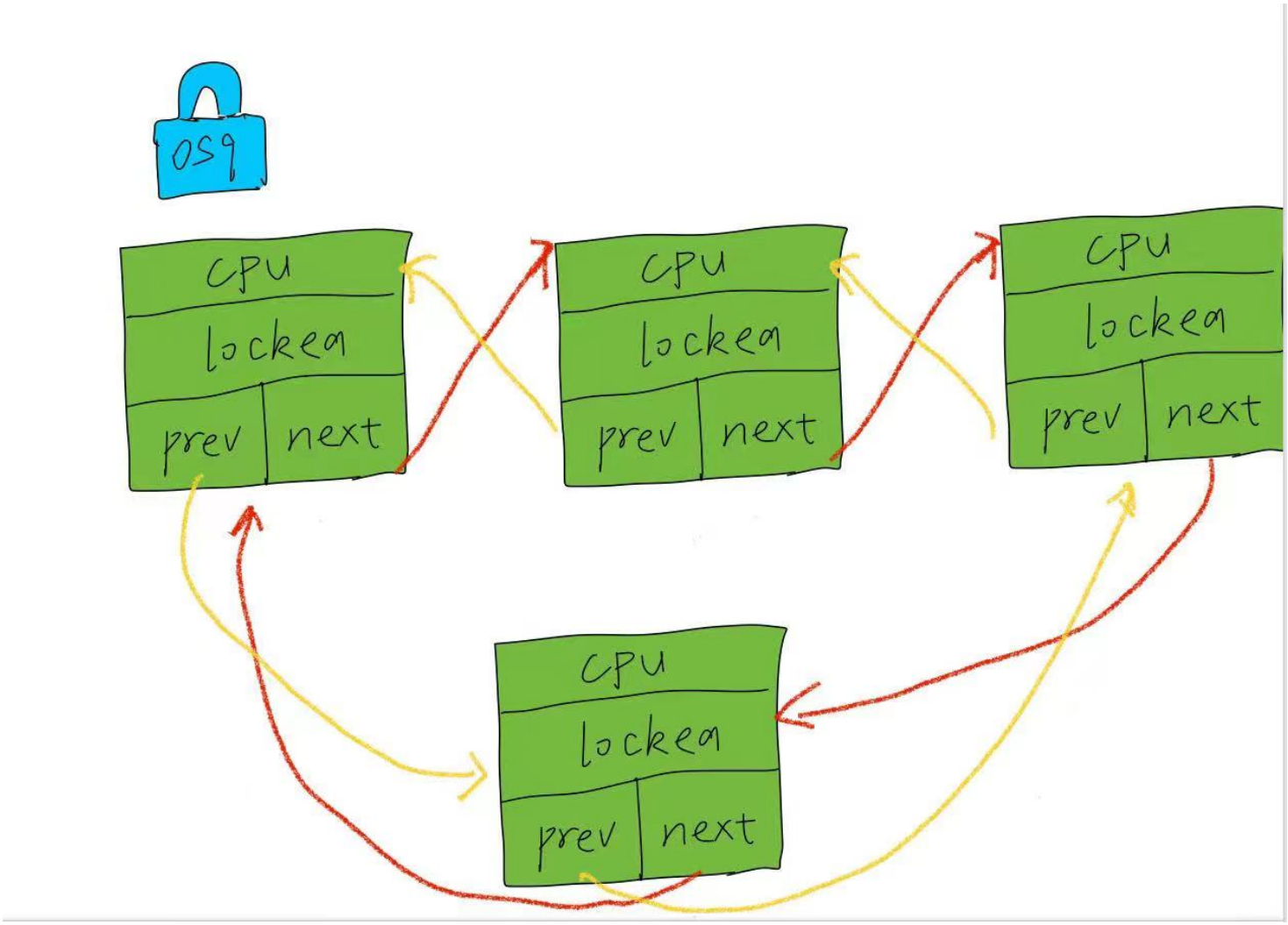
根据上面的场景，在汽车进入收费站之前首先会卡在快车道的卡口出，也就是osq锁（optimistic_spin_queue 乐观自旋队列),所以我们先看osq锁的实现。

osq锁的实现

1. osq锁的数据结构

```
struct optimistic_spin_node {
    struct optimistic_spin_node *next, *prev;
    int locked;  //lock的值为1代表锁被占有
    int cpu;     //将cpu编号进行编码
};

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;  //tail的值表示当前等待队列最后一个节点所在CPU编号
};
```

各节点间关系如图：



每个链表节点中有个数据是 cpu 代表进程所在任务的cpu编号映射（不是系统的CPU编号），**当这个cpu变量的值为0时，在整个osq锁机制中代表着这个任务所在的CPU执行别的任务去了**（即进程处在休眠或已经结束）。这个cpu的值等于0的时候有特殊意义，因此就不能直接使用系统CPU编号（系统CPU编号从0开始）。具体的映射关系就是`struct optimistic_spin_node`中的cpu值刚好比系统CPU编号大1。
对应的cpu值与系统CPU编号映射关系函数如下：

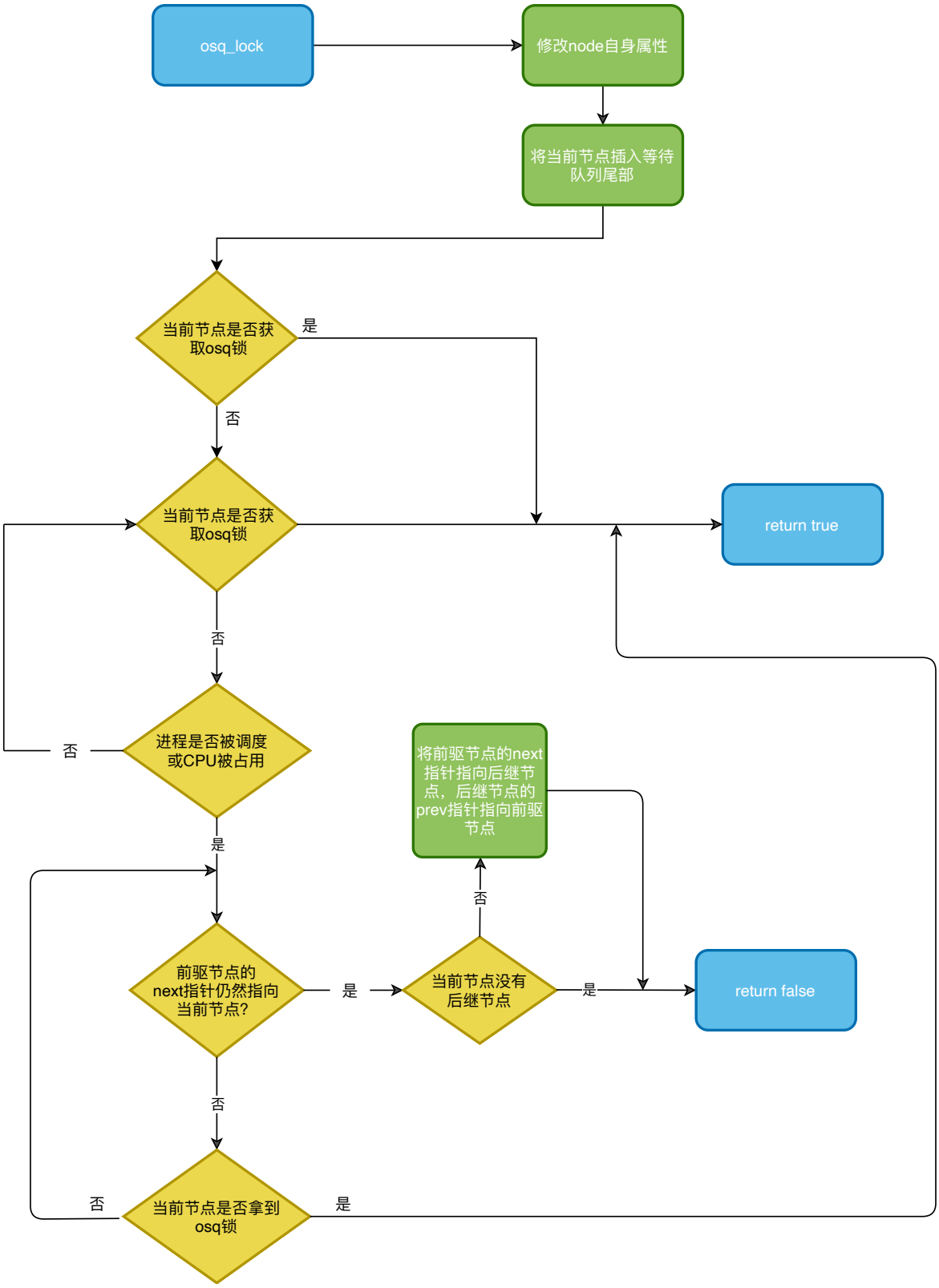
```
//由CPU编号获得在 osq_lock 中对应的CPU编号
static inline int encode_cpu(int cpu_nr)
{
    return cpu_nr + 1;
}

//将 函数 encode_cpu 编码出来的CPU编号解码成系统对应的CPU编号
static inline struct optimistic_spin_node *decode_cpu(int encoded_cpu_val)
{
    int cpu_nr = encoded_cpu_val - 1;

    return per_cpu_ptr(&osq_node, cpu_nr);
}
```

另外，还有一个通过从`struct optimistic_spin_node`节点映射到系统CPU的函数。

```
//由 MCS锁中的 optimistic_spin_queue 结构获得系统内的CPU编号
static inline int node_cpu(struct optimistic_spin_node *node)
{
    return node->cpu - 1;
}
```

2.osq锁的加锁实现

osq锁进行加锁的函数逻辑如下：



具体代码实现如下：

osq_lock函数的实现可以分为四个部分：

1. 更改node结点信息
2. 尝试直接拿取osq锁
3. 拿锁失败，加入等待队列循环等待
4. 等待过程中进程被调度执行对应方法

第一部分：更改node结点信息

```
    //将 locked 赋值为0表示当前节点并未持有锁
    node->locked = 0;
    node->next = NULL; //后继结点为空
    node->cpu = curr;  //CPU编号就是当前CPU编号
```

第二部分：尝试直接拿取osq锁

```
    //如果赋值前 lock->tail 的值为0，那么这时当前进程可以直接获得锁，即申请锁成功
    //这里的意思是如果 lock->tail 的值为0，那么代表锁的前一个持有者已经执行完毕
    if (old == OSQ_UNLOCKED_VAL)
        return true;
```

这里宏`OSQ_UNLOCKED_VAL`定义为0，old是等待队列里最后一个任务所在CPU经过编码后的编号，前面已经提到当编码后的CPU编号为0则代表对应任务已经不在运行状态了（被调度或运行完成），那么 **当`tail`的值为0，则代表没有任务占用osq锁。** 这个时候当前任务就可以直接拿到osq锁。

对于`old`的赋值，利用了`atmotic_xchg`函数将等待队列最后一个结点的CPU编号改为当前申请osq锁结点对应的CPU编号（就是将当前结点插入等待队列的末尾）同时获得插入结点前等待队列末尾的结点对应CPU编号（这个编号的值赋给了`old`变量）。代码如下：

```
    //使用原子操作将 当前CPU编码结果赋值给 lock 中的原子变量 tail ，
    // 并返回赋值前 lock->tail 的值，将其赋给 old
    old = atomic_xcchg(&lock->tail, curr);
```

第三部分：尝试拿锁失败，自旋等待直到申请到锁
观察循环体的循环条件，当申请到锁的时候结束循环，执行循环体下面的返回`true`语句。

```
   //不停循环，直到当前节点获得锁
    while (!READ_ONCE(node->locked)) {

        //如果进程被调度或睡眠，跳出当前循环，进入 unqueue
        if (need_resched() || vcpu_is_preempted(node_cpu(node->prev)))
            goto unqueue;

        cpu_relax();
    }
    return true;
```

（所谓自旋就是在循环忙等，对应上面的循环体结构）

第四部分：自旋时被调度，进入unquene，退出等待队列

> 注意这里的言外之意就是等待队列中的进程都在循环忙等，这个队列将因为申请了osq锁却拿不到只能循环等待的进程串起来了，当某个节点不满足循环等待的条件（还未拿到锁且没有被调度、抢占）的时候就从队列的移除

退出等待队列的过程，先查看进程在被调度之后和 **调度前的前驱节点** 相对关系是否发生变化，如果变化了，更新前驱结点，尝试获取锁；最后如果当前结点后面还有有结点的话就把后面的结点拼接到前驱结点后面，完成自身从等待队列的移除。

循环更新前驱结点，并尝试获取锁

```
    for (;;) {

        //如果当前节点和其前驱节点的关系没有改变 将当前节点脱离链表并跳出循环
```

```
        if (prev->next == node &&
            cmpxchg(&prev->next, node, NULL) == node)
            break;


        //如果此时获得了锁就返回 true
        //前面已经判断过一次了为什么还要判断一下此时有没有拿到锁呢?
        //因为这里 node 的前继节点已经被改变则证明前一时刻锁被别的进程拿走了,
        //这里需要再判断一下是不是刚好别的进程退出临界区并释放了锁，被当前进程拿到
        if (smp_load_acquire(&node->locked))
            return true;

        cpu_relax();


        //程序运行到这里 prev 已经不是 node 的前驱节点，需要重新获取
        prev = READ_ONCE(node->prev);
    }
```

既然已经决定要从等待队列中移除为什么不直接执行链表移除结点的操作还要加一个循环呢?
首先，我们要了解为什么会进入到`unqueue`分支当中，就是因为在循环等待的过程中进程被调度了，在CPU运行其他进程的时候发生了什么是无法预料的，因此调度期间 **锁持有者发生了变更并且当前结点是锁持有者的后继节点**，这种情况也是有可能发生的，这样就需要接着循环等待直到拿到锁。

循环结束后，如果没有后继结点了直接返回`false`，否则还要处理后继结点。

```
    //由 osq_wait_next 代码可知，返回的 next 是 node 的后继节点
    next = osq_wait_next(lock, node, prev);
    //如果 next 为空，代表被删除时 node 已经是最后一个节点,
       //直接返回 false,没拿到锁
    if (!next)
        return false;




    //如果 next 不为空，将将继节点的 next 指向后继节点,
      //后继节点的 prev 指向前继节点。 返回false
    WRITE_ONCE(next->prev, prev);
    WRITE_ONCE(prev->next, next);

    return false;
```

至此，osq锁的加锁函数已经实现完毕，完整代码如下：

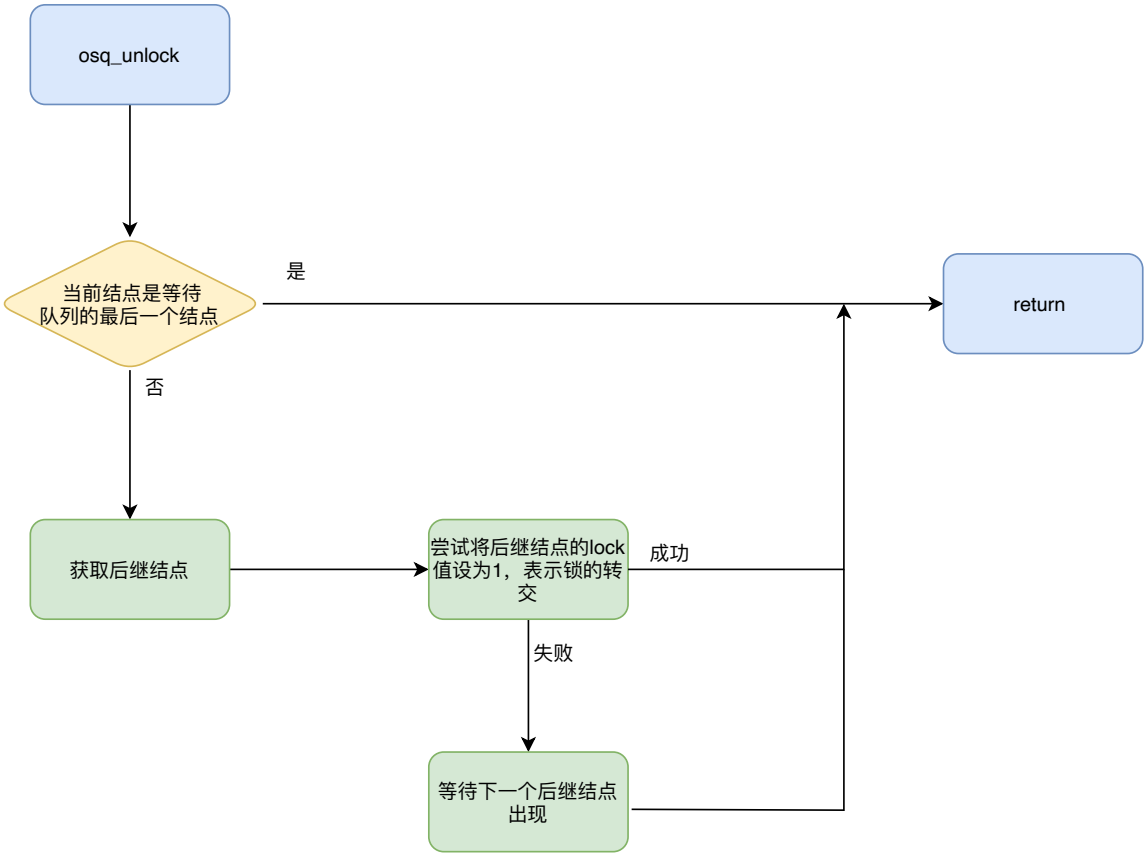
```
bool osq_lock(struct optimistic_spin_queue *lock)
{
    //当前节点的CPU编号
    struct optimistic_spin_node *node = this_cpu_ptr(&osq_node);

    struct optimistic_spin_node *prev, *next;
    //将处理当前进程的CPU编号进行编码
    int curr = encode_cpu(smp_processor_id());
    int old;
    //将 locked 赋值为0表示当前节点并未持有锁
    node->locked = 0;
    node->next = NULL;
    node->cpu = curr;
```

```
    //使用原子操作将 当前CPU编码结果赋值给 lock 中的原子变量 tail ，
      // 并返回赋值前 lock->tail 的值，将其赋给 old
    old = atomic_xcchg(&lock->tail, curr);

    //如果赋值前 lock->tail 的值为0，那么这时当前进程可以直接获得锁，即申请锁成功
    //这里的意思是如果 lock->tail 的值为0，那么代表锁的前一个持有者已经执行完毕
    if (old == OSQ_UNLOCKED_VAL)
        return true;
//以下为赋值前 lock->tail 的值不为0的情况
//lock->tail不为0代表锁的持有者还在占用CPU运行
    //执行链表的节点插入操作（将当前节点插入链表）
    prev = decode_cpu(old);
    node->prev = prev;
    WRITE_ONCE(prev->next, node);



    //不停循环，直到当前节点获得锁
    while (!READ_ONCE(node->locked)) {


        //如果进程被调度或睡眠，跳出当前循环，进入 unqueue
        if (need_resched() || vcpu_is_preempted(node_cpu(node->prev)))
            goto unqueue;

        cpu_relax();
    }
    return true;

unqueue:


    for (;;) {

        //如果当前节点和其前驱节点的关系没有改变 将当前节点脱离链表并跳出循环
        if (prev->next == node &&
            cmpxchg(&prev->next, node, NULL) == node)
            break;


        //如果此时获得了锁就返回 true
        //前面已经判断过一次了为什么还要判断一下此时有没有拿到锁呢?
        //因为这里 node 的前继节点已经被改变则证明前一时刻锁被别的进程拿走了，
        //这里需要再判断一下是不是刚好别的进程退出临界区并释放了锁，被当前进程拿到
        if (smp_load_acquire(&node->locked))
            return true;

        cpu_relax();


        //程序运行到这里 prev 已经不是 node 的前驱节点，需要重新获取
        prev = READ_ONCE(node->prev);
    }


    //由 osq_wait_next 代码可知，返回的 next 是 node 的后继节点
    next = osq_wait_next(lock, node, prev);
    //如果 next 为空，代表被删除时 node 已经是最后一个节点，
       //直接返回 false,没拿到锁
    if (!next)
```

```
        return false;



    //如果 next 不为空，将将继节点的 next 指向后继节点，
      //后继节点的 prev 指向前继节点。 返回false
    WRITE_ONCE(next->prev, prev);
    WRITE_ONCE(prev->next, next);

    return false;
}
```

3. osq锁的解锁实现

osq锁的解锁过程逻辑简单，如下图：



判断当前节点是否是等待队列的队尾结点时，使用原子操作`atomic_cmpxchg_release`,如果判断结果是真的话同时将等待队列的`tail`值改为0，表示没有任务持有锁，后续申请者可以直接获得。

```
    //如果当前节点是最后一个节点，直接将 lock->tail 设置为0 表示锁空闲，返回函数
    if (likely(atomic_cmpxchg_release(&lock->tail, curr,
                      OSQ_UNLOCKED_VAL) == curr))
        return;
```

当前结点不是队尾结点的话尝试转交锁，转交成功直接返回，放锁过程结束，转交不成功需要等待下一个后继结点的出现，将锁转交给下一个后继结点，结束放锁。

```
    //如果当前节点不是最后一个节点，通过原子操作获取后继节点
    //将后继节点 locked 设为 1，表示持有锁
    node = this_cpu_ptr(&osq_node);
    next = xchg(&node->next, NULL);
    if (next) {
```

```
        WRITE_ONCE(next->locked, 1);
        return;
    }
    //获取后继节点失败，等待下一个后继节点出现
    next = osq_wait_next(lock, node, NULL);
    if (next)
        WRITE_ONCE(next->locked, 1);
```

之所以说是 **等待** 下一个后继结点，是因为在`osq_wait_next`函数中有一个死循环，跳出循环的条件之一就是 **成功获取当前结点的后继结点** 。下面是`osq_wait_next`函数中的循环体：

```
//如果prev 的 cpu 编号存在，将其赋值给 old。否则将 0 赋给 old
old = prev ? prev->cpu : OSQ_UNLOCKED_VAL;

//lock->tail==curr 代表当前节点是链表的最后一个节点，
//以下循环主要是为了实现：
//1、当前节点为最后一个节点时，删除该节点，返回NULL
//2、当前节点存在后继节点时，解除前驱节点对当前节点的指针
//   找到当前节点的后继节点并返回
for (;;) {
    //如果 lock->tail 为当前进程cpu 编号，
    //将前驱节点的 cpu编号赋给 lock->tail,跳出循环

    if (atomic_read(&lock->tail) == curr &&
          atomic_cmpxchg_acquire(&lock->tail, curr, old) == curr) {

        break;
    }
    //如果当前节点存在后继节点，将 node->next 赋值为null，跳出循环
    if (node->next) {
        next = xchg(&node->next, NULL);
        if (next)
            break;
    }

        cpu_relax();
}
```

(`osq_wait_next`函数的整体实现略去，完成代码在文件`osq_lock.c`中)

`osq_unlock`函数完整实现如下：

```
void osq_unlock(struct optimistic_spin_queue *lock)
{
    struct optimistic_spin_node *node, *next;
    int curr = encode_cpu(smp_processor_id());

    //如果当前节点是最后一个节点，直接将 lock->tail 设置为0 表示锁空闲，返回函数
    if (likely(atomic_cmpxchg_release(&lock->tail, curr,
                      OSQ_UNLOCKED_VAL) == curr))
        return;

    //如果当前节点不是最后一个节点，通过原子操作获取后继节点
    //将后继节点 locked 设为 1，表示持有锁
    node = this_cpu_ptr(&osq_node);
    next = xchg(&node->next, NULL);
    if (next) {
        WRITE_ONCE(next->locked, 1);
        return;
    }
    //获取后继节点失败，等待下一个后继节点出现
```

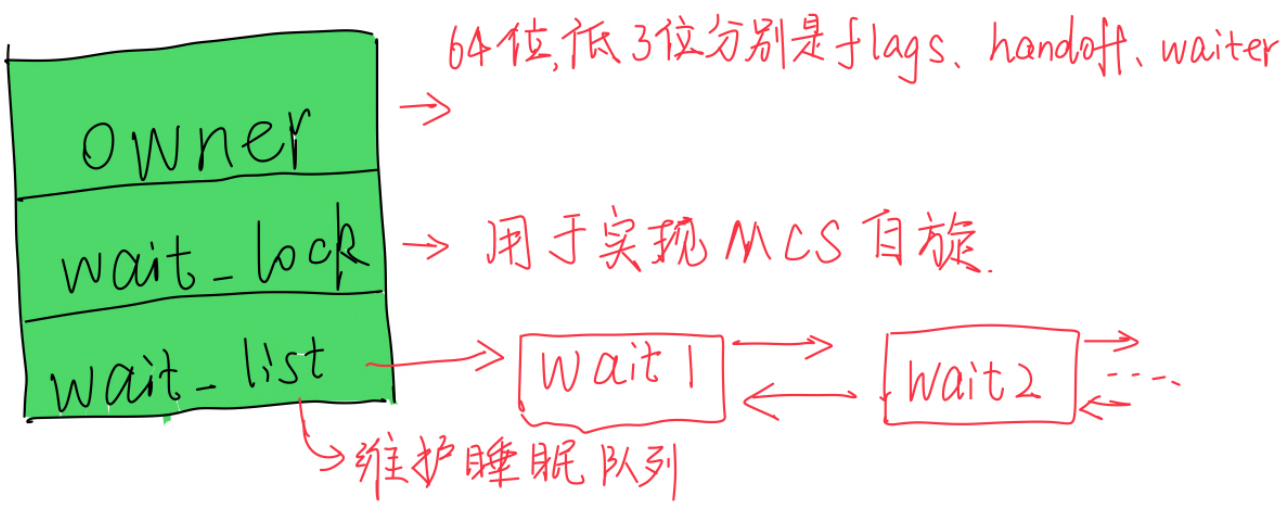
```
    next = osq_wait_next(lock, node, NULL);
    if (next)
        WRITE_ONCE(next->locked, 1);
}
```

osq锁的作用

将申请同一个osq锁的任务排成队列并各自循环等待，以链表形式维护，队列中的元素可以通过指针联系前后结点。
回顾开头的场景，osq锁的作用相当于将所有快车道尽头的关卡串在一起了，同一时刻只有一个关卡是开的（对应osq锁的持有者）其他关卡都是关的，关卡前面的司机都在等待关卡打开（循环等待）。关于osq锁的作用，后面会在mutex锁的实现中进一步讨论。

Mutex锁的实现

(1) mutex锁的数据结构



(2) mutex 的初始化

如果定义了宏 `CONFIG_MUTEX_SPIN_ON_OWNER` 即使用乐观自旋队列的话除了将mutex的三个属性初始化还需要初始化osq锁。

> owenr初始化为64位全0
> wait_lock 自旋锁的初始化
> wait_list 初始化睡眠链表的表头

```
void
__mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key)
{
    //将owner初始化为0，同时初始化 自旋锁 wait_lock 和链表 wait_list
    atomic_long_set(&lock->owner, 0);
    spin_lock_init(&lock->wait_lock);
    //wait_lock 的作用在于保护等待队列
    INIT_LIST_HEAD(&lock->wait_list);
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
```

```
    osq_lock_init(&lock->osq);
#endif

    debug_mutex_init(lock, name, key);
}
EXPORT_SYMBOL(__mutex_init);
```

(3) mutex 加锁实现

在阅读`mutex.c`文件的时候发现，只有少数几个函数前面没有关键字`static`修饰，查询相关资料后表示mutex锁只能通过其提供的接口使用，也就是使用mutex锁的时候只能调用这里面没有关键字`static`修饰的函数。包括

```
void  __mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key);
void __sched mutex_lock(struct mutex *lock);
void __sched mutex_unlock(struct mutex *lock);
void __sched ww_mutex_unlock(struct ww_mutex *lock);
int __sched mutex_trylock(struct mutex *lock);
```

---

1. `mutex_trylock` 与 `__mutex_trylock` 解析

`mutex_trylock`函数内部调用了`__mutex_trylock`函数，然后返回`__mutex_trylock`函数的返回值。 **可以直接理解为向外部提供了一个调用内部函数`__mutex_terlock`的方法，方法名是`mutex_trylock`。**

`mutex_trylock`的源码如下：

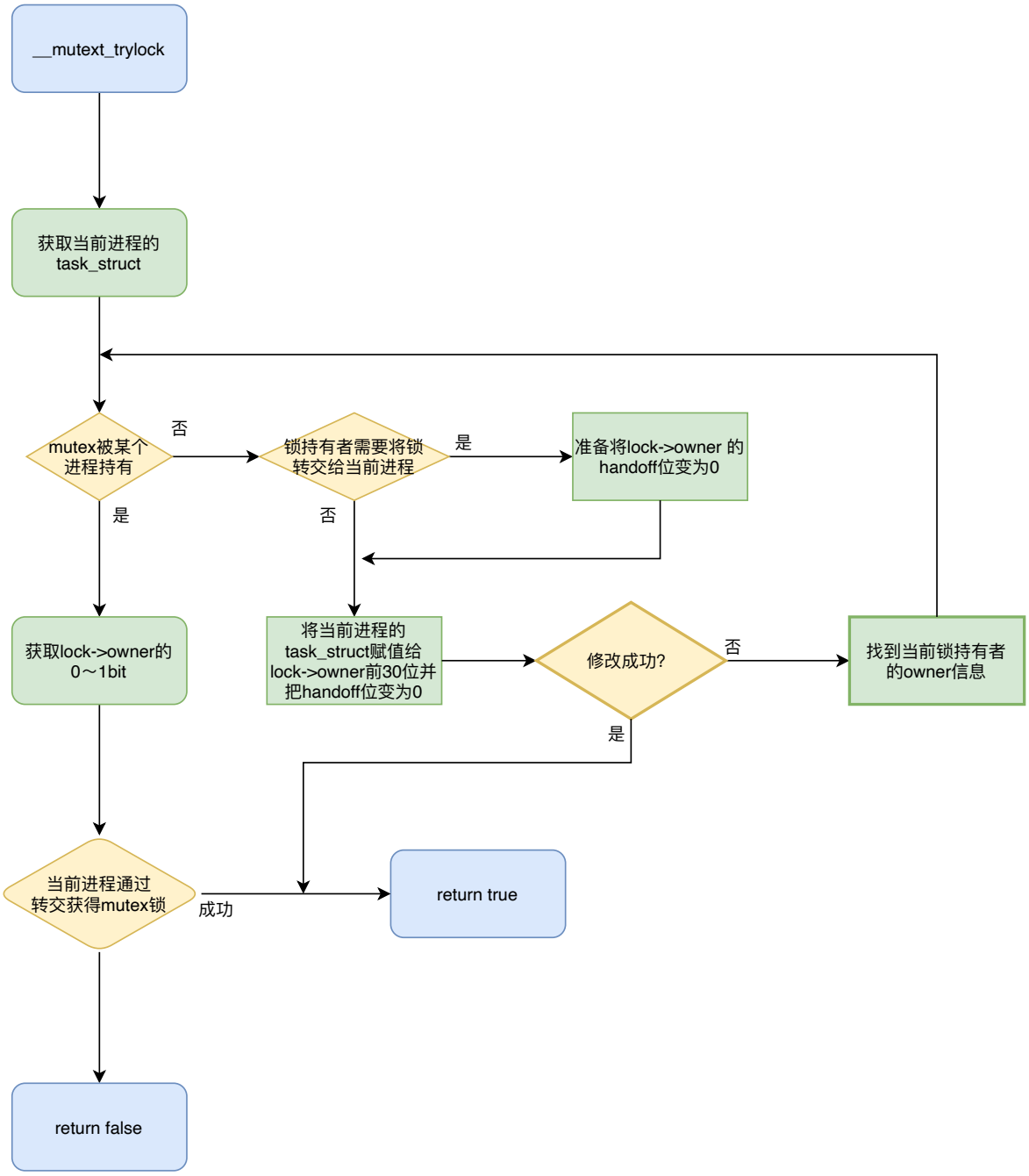
```
int __sched mutex_trylock(struct mutex *lock)
{
    bool locked = __mutex_trylock(lock, false);

    if (locked)
    // mutex_acquire 没用
        mutex_acquire(&lock->dep_map, 0, 1, _RET_IP_);

    return locked;
}
```

函数`mutex_acquire`对于使用锁的过程来说是没有意义的，其映射关系如下

> mutex_acquire -> lock_acquire_exclusive -> lock_acquire -> do{ } while(0)

```
#define mutex_acquire(l, s, t, i)       lock_acquire_exclusive(l, s, t, NULL, i)
#define lock_acquire_exclusive(l, s, t, n, i)       lock_acquire(l, s, t, 0, 1, n, i)
#define lock_acquire(l, s, t, r, c, n, i)   do { } while (0)
```

- `__mutex_trylock`函数尝试获取锁，获取不到的话就退出函数返回`false`，获取到锁就返回`true`，其逻辑如下：



> 在确定了锁持有者需要将锁转交给当前进程之后，是 **准备** 将 `lock->owner` 的 `handoff` 位变为0,而不是立马改变。原因就是在具体的代码实现中，并不是每次涉及到 `lock->owner` 就直接调用 `lock->owner` 的，而是定义了一个 `unsigned long` 类型的变量 `owner` ，再将 `lock->owner` 的值赋给它，需要比较 `lock->owner` 值的时候就直接比较 `owner` 的值，在给 `lock->owner` 赋值之前，先将想要的值赋给 `owner` 最后再将 `owner` 赋值给 `lock->owner` 。但是这样也存在一个问题，就是比较 `owner` 的时候，他的值 **不一定就是当前 `lock->owner`** 的值，因此也就需要及时的找到 `lock->owner` 的值并更新 `owner` 。

首先定义了两个 `unsigned long` 变量，分别存储当前任务的 `task_struct` 和 `lock->owner` 两个值。

```
    //获取当前锁持有者状态
    unsigned long owner, curr = (unsigned long)current;
    //提取出 lock->owner
    owner = atomic_long_read(&lock->owner);
```

接下来进入一个循环体，进入循环后，如果mutex锁有持有者的话，查看持有者是不是当前进程，是就直接返回 `true` ，不是就返回 `false` ;如果没有持有者的话代表 `lock->owner` 前30位为0，将当前任务的 `task_struct` 赋值给 `lock->owner` 前30位，倒数第二位，也就是 `handoff` 位，变为0，最后一位保持不变。更改成功的话返回 `true` ,否则返回 `false` 。

```
for (;;) {
    //将owner的最后两位提取出来给 flag
    unsigned long old, flags = __owner_flags(owner);
    //如果 owner 的前30位存在，
    // __owner_task 函数返回值不为空，即owner前30位不为0时
    //进入下面的代码块
    if (__owner_task(owner)) {
        //当锁需要转交 且 owner 的前30位与 current 位相同表示获取锁成功
        //owner前30位与current相同代表锁持有者是当前进程
        if (handoff && unlikely(__owner_task(owner) == current)) {

            //此时已经成功获得锁，使用内存屏障让所有竞争者知道锁已经转交
            smp_mb();
            return true;
        }
        //当不需要转交或锁被其他任务持有，返回false
        return false;
    }

//注意：代码能运行到这里，肯定是没有进入上面的 IF 判断体的
//也就是锁还没有被持有

//下面的代码目的是，在被唤醒线程通过转交获取锁的时候，需要将owner的转交位重新置0
    //如果锁需要转交，将flags的倒数第二位置为0
    if (handoff)
        flags &= ~MUTEX_FLAG_HANDOFF;

    //利用原子操作再次确认这段时间内锁没有被其他人拿走
    //转交锁之前，修改锁的 flag 标识位
    old = atomic_long_cmpxchg_acquire(&lock->owner, owner, curr | flags);
    if (old == owner)
        return true;
    owner = old;
}
```

2.mutex_lock 解析

对于`mutex_lock`就是先尝试快速获取锁，如果获取不到的话就慢速获取锁。前面提到的场景中，司机发现旁边都没有车辆，就直接开上高速公路了，如果还有其他车辆的话就需要等待了，说得就是`mutex_lock`的逻辑。

```
//锁的获取
void __sched mutex_lock(struct mutex *lock)
{
    //可能会睡眠
    might_sleep();

    //尝试获取快道失败就获取慢道
//这里的逻辑从 mutex锁 整体来看，先尝试直接获取锁，
            //拿不到的话就尝试自旋等待、睡眠等待
    if (!__mutex_trylock_fast(lock))
        __mutex_lock_slowpath(lock);
}
```

函数`__mutex_trylock_fast`就是司机旁边没有任何其他车辆的情况，对应的就是`lock->owner`的64位全为0，没有人占用锁也没有人竞争锁，当前任务可以直接拿到锁。‘

```
//直接加锁
static __always_inline bool __mutex_trylock_fast(struct mutex *lock)
```

```
{
    unsigned long curr = (unsigned long)current;
    //如果 lock->owner 全为0，代表锁没有持有者，可以直接获取
    if (!atomic_long_cmpxchg_acquire(&lock->owner, 0UL, curr))
        return true;

    return false;
}
```

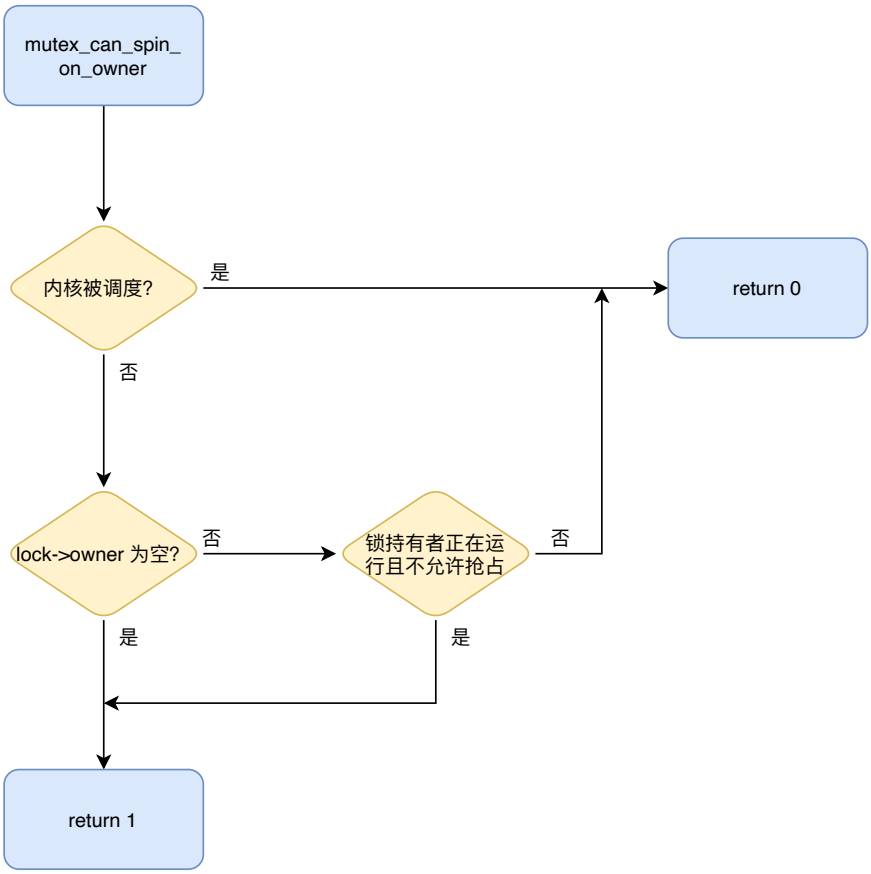
慢速获得锁的过程，`__mutex_lock_slowpath`只是调用了另外一个函数`__mutex_lock_common`。

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

在介绍`__mutex_lock_commom`之前需要补充介绍几个函数。

关于`__mutex_lock_common`函数的补充介绍

- `mutex_can_spin_on_owner`解析
  这个函数只有在内核不允许调度、`lock->owner`（锁持有者进程）正在运行且不允许抢占的情况下才会返回1，否则返回0。
  执行逻辑如下：



根据上面的逻辑可以知道，这个函数的作用就是 **通过查看锁持有者的状态来判断其他申请锁的进程能否达到自旋等待的条件。** 因为如果锁持有者被调度的话，短时间内无法释放锁，其他进程的自旋消耗的CPU资源比进程的上下文切换代价还要大，那么就会将自旋等待的进程切换到睡眠状态。

在函数实现内部，对锁持有者信息进行查看的时候还是用了`rcu_read_lock`读者锁,确保对相关信息判断的正确性（进行判

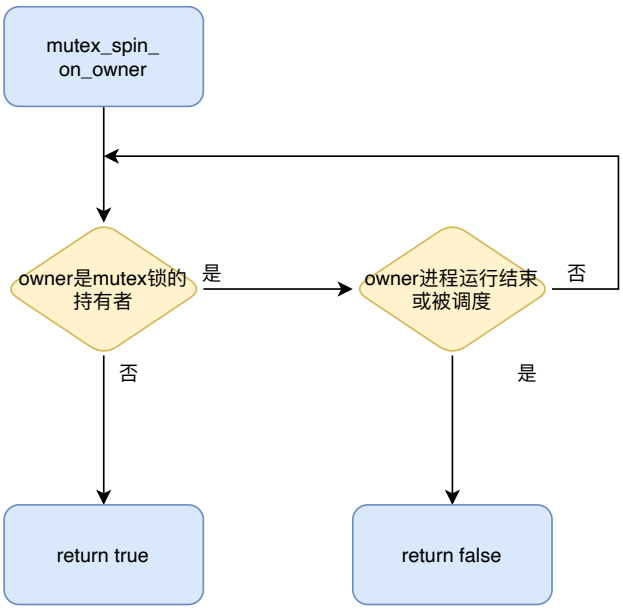
断的过程中锁不会被其他进程拿走或修改owner信息）。
函数实现代码如下：

```
// mutex 锁被持有且持有者正在执行，不允许抢占、调度时返回 1，否则返回 0
//返回1时，lock->owner可以自旋，返回0就不可以自旋
static inline int mutex_can_spin_on_owner(struct mutex *lock)
{
    //lock->owner 指向该进程的 task_struct结构
    // task_struct->on cpu 为1时表示锁持有者正在临界区运行，
    //当锁被锁持有者释放后， lock->owner 为 NULL
    struct task_struct *owner;
    int retval = 1;
    //内核被调度，返回0
    if (need_resched())
        return 0;

//使用rcu_lock来创建临界区是为了 owner 指针所指的 task_struct
  // 不会因为你进程被杀而导致访问owner指针出错
  //RCU指针可以保护 task_struct 结构在临界区不被释放
    rcu_read_lock();
    owner = __mutex_owner(lock);

    if (owner)
    //当前锁持有者正在执行且关闭了内核抢占时，retval的值为1，否则为0
        retval = owner->on_cpu && !vcpu_is_preempted(task_cpu(owner));
    rcu_read_unlock();

    //如果返回0，说明当前锁持有者并不在临界区执行，或内核可以被抢占，或锁空闲
    return retval;
}
```

- `mutex_spin_on_owner` 解析
  函数`mutex_spin_on_owner` 使用 `rcu_read_lock` 读者锁进入循环，不断比较`lock->owner` 和 `owner`是否相等，相等继续循环，不等直接返回 `true`,如果循环过程中发生调度或者 `owner` 执行结束返回 `false` 。

> 该函数的 **目的是让进程`owenr` 在`mutex` 锁上自旋**，所以传入该函数的参数`owner`应该保证其大概率是`mutex`锁的持有者，否则函数直接返回，这样做也没有什么意义（只要在mutex锁的实现中没有意义）。

函数运行逻辑如下：



具体实现如下：

```
static noinline
//锁 被其他 owner 持有时，直接返回true
//被owner持有时，不断循环，等待 owner 被抢占调度，返回 false（该等待过程被加锁了）
//从 mutex锁 的机制来看，当mutex被其他进程持有时，直接返回true；
    //锁被本进程持有时，不断循环检测当前进程是否被调度
//只有在owner持有锁的时候放好被调度了才户返回false，不然都是true
bool mutex_spin_on_owner(struct mutex *lock, struct task_struct *owner)
{
    bool ret = true;
    //获得rcu读锁，进入临界区
    rcu_read_lock();
    // __mutex_owner 的作用是通过 lock 获得锁的持有者
    //如果 mutex 锁被 owner 持有，不断自旋等待
    //除非持有者被调度，返回 false
    while (__mutex_owner(lock) == owner) {

        barrier();

        //当锁持有者进程在睡眠、被调度、内核被抢占时返回 false
        if (!owner->on_cpu || need_resched() ||
                vcpu_is_preempted(task_cpu(owner))) {
            ret = false;
            break;
        }

        cpu_relax();
    }
    rcu_read_unlock();

    return ret;
}
```

- `mutex_optimistic_spin` 函数解析

对于进入 `mutex_optimistic_spin` 函数的进程可能有以下情况：程序运行刚开始的时候，等待队列为空、程序正常运行中，等待队列不为空。

> 程序刚开始的时候， `osq_lock` 锁只是被初始化了，并没有被使用过，所以对于进程刚开始的时候进入乐观自旋的进程应该先去获取 `osq_lock` 如果不去拿 `osq_lock` 锁的话那么后续的进程也不会去拿这个锁。这会导致一个严重的情况，相当于丢失了 `osa_lock` ,那么所有申请 `mutex_lock` 的进程都会在 **全局变量下自旋， 导致大量缓存页颠簸情况出现**，整体效率大打折扣。言外之意就是在 `mutex_lock` 中使用 `osq_lock` 锁是为了将申请 `mutex_lock` 锁的绝大部分进程放在自身所在CPU上自旋，而不是在全局环境下自旋，只有一个拿到 `osq_lock` 锁的进程在全局环境下自旋。这样就减少了缓存页颠簸情况的出现。

- 等待队列为空时判断锁持有者是否被调度且尝试获取锁

```
    //当前进程不是等待进程时
    //注意：这里的 waiter 与owner的最后一位代表的 waiter不同
    if (!waiter) {

        //mutex锁 拥有者在睡眠或被调度时，不能自旋等待
        if (!mutex_can_spin_on_owner(lock))
            goto fail;

        //获取 osq 锁失败，跳转 fail
        if (!osq_lock(&lock->osq))
            goto fail;
    }
```

- 在程序运行中进入该函数的进程，如果等待队列不为空，首先避免与其他的 `mutex_lock` 和进程形成死锁；其次判断锁

持有者是否非空。如果锁持有者非空的话判断当前线程是否因为转交而获取锁、锁持有者是否被调度。最后尝试获取锁。这也是函数在进入循环后做的三件事。

1. 避免死锁发生

```
//乐观自旋做的第一件事:
    //避免发生死锁
        if (use_ww_ctx && ww_ctx->acquired > 0) {
            struct ww_mutex *ww;

            //由 lock 得到 struct ww_mutex
            ww = container_of(lock, struct ww_mutex, base);

            // ww 存在上下文就跳转到 fail_unlock
            if (READ_ONCE(ww->ctx))
                goto fail_unlock;
        }
```

2. 锁持有者不为空的话判断当前进程是否因为转交而获得锁、锁持有者是否被调度。

```
//乐观自旋做的第二件事:
    //判断锁被持有的情况下;
    //1、判断持有者是否是当前进程且有任务在等待队列中
    //2、判断锁持有者是否被调度进入睡眠
        //获取锁持有者信息
        owner = __mutex_owner(lock);
        if (owner) {
            //有任务正在等待且当前任务持有锁 退出乐观自旋
            //这里 owner==task 判断成功的意思应该是mutex锁的前一个持有者
                //将锁给转交过来的 。
            //可以理解为是前一个持有者将 owner 设置为 task 的
            if (waiter && owner == task) {
                smp_mb();
                break;
            }
        //到这里要么是没有等待任务，要么是锁持有者不是当前任务
            //当锁被owner持有且持有者最终被调度时 跳转到 fail_unlock
            if (!mutex_spin_on_owner(lock, owner))
                //进入这个 if 结构只可能是owner是lock的持有者且被调度了
                //锁持有者被调度进入睡眠，无法解锁
                goto fail_unlock;
        }
```

3. 尝试获取锁

```
//乐观自旋做的第三件事:
    //尝试获取 mutex锁
        if (__mutex_trylock(lock, waiter))
            break;
```

整个函数执行逻辑如下：


mutex_optimistic
_spin
获取当前进程的
task_struct结构
等待队列为空?
否
避免发生死锁
是
获取lock->owner信息
mutex->owner可以
自旋等待
否
是
owner非空
否
是
尝试获取osq锁
失败
成功
释放mutex的osq锁
有其他
任务抢占CPU
否
是
等待队列非空且当
前任务是锁持有者
否
是
等待队列非空?
是
否
保存当前运行状态
锁持有者进程
运行中被调度
是
否
禁止内核抢占
return false
失败
尝试获取mutex锁
成功
return true
将mutex锁的
osq锁解锁
否
等待队列非空?

完整的代码实现如下：

```
//乐观自旋锁
//该函数返回 true 时，代表结束自旋且拿到mutex锁
static bool mutex_optimistic_spin(struct mutex *lock,
                  struct ww_acquire_ctx *ww_ctx,
                  const bool use_ww_ctx, const bool waiter)
{
    //当前进程的 task_struct 结构
    struct task_struct *task = current;
    //当前进程不是等待进程时
    //注意：这里的 waiter 与owner的最后一位代表的 waiter不同
    if (!waiter) {

        //mutex锁 拥有者在睡眠或被调度时，不能自旋等待
        if (!mutex_can_spin_on_owner(lock))
            goto fail;

        //获取 osq 锁失败，跳转 fail
        if (!osq_lock(&lock->osq))
            goto fail;
    }
//如果等待队列不为空，或者等待队列为空时锁拥有者正在运行且无法被打断且获得 osq 锁
    //进入下面的循环，也就是乐观自旋，在乐观自旋中不断循环尝试三件事
    for (;;) {
        struct task_struct *owner;
//乐观自旋做的第一件事：
    //避免发生死锁
        if (use_ww_ctx && ww_ctx->acquired > 0) {
            struct ww_mutex *ww;

            //由 lock 得到 struct ww_mutex
            ww = container_of(lock, struct ww_mutex, base);

            // ww 存在上下文就跳转到 fail_unlock
            if (READ_ONCE(ww->ctx))
                goto fail_unlock;
        }


//乐观自旋做的第二件事：
    //判断锁被持有的情况下；
    //1、判断持有者是否是当前进程且有任务在等待队列中
    //2、判断锁持有者是否被调度进入睡眠
        //获取锁持有者信息
        owner = __mutex_owner(lock);
        if (owner) {
            //有任务正在等待且当前任务持有锁 退出乐观自旋
            //这里 owner==task 判断成功的意思应该是mutex锁的前一个持有者
                //将锁给转交过来的 。
            //可以理解为是前一个持有者将 owner 设置为 task 的
            if (waiter && owner == task) {
                smp_mb(); /* ACQUIRE */
                break;
            }
        //到这里要么是没有等待任务，要么是锁持有者不是当前任务
            //当锁被owner持有且持有者最终被调度时 跳转到 fail_unlock
            if (!mutex_spin_on_owner(lock, owner))
                //进入这个 if 结构只可能是owner是lock的持有者且被调度了
                //锁持有者被调度进入睡眠，无法解锁
                goto fail_unlock;
```

```
        }

        //当 waiter 成功拿到锁时跳出循环（也是退出乐观自旋）
//乐观自旋做的第三件事：
    //尝试获取 mutex锁
        if (__mutex_trylock(lock, waiter))
            break;

        cpu_relax();
    }
//如果代码能够运行到这里，代表着 owner 现在已经是 mutex锁 的持有者
    //如果没有等待任务，释放 osq锁
    if (!waiter)
        osq_unlock(&lock->osq);
//乐观自旋函数一旦返回true，代表已经结束自旋并且获得锁了
    return true;


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

fail:

    //如果其他任务抢占CPU
    if (need_resched()) {

        //保存当前状态
        __set_current_state(TASK_RUNNING);
        //禁止内核抢占
        schedule_preempt_disabled();
    }

    return false;
}
#else
static bool mutex_optimistic_spin(struct mutex *lock,
                  struct ww_acquire_ctx *ww_ctx,
                  const bool use_ww_ctx, const bool waiter)
{
    return false;
}
```

回到 `__mutex_lock_common` 函数

`__mutex_lock_common` 函数主要分为三块：

> 1. 进行死锁避免的相关处理
> 2. 尝试直接获取锁，获取不到进入乐观自旋
> 3. 乐观自旋失败的话进入睡眠等待队列

- 死锁避免

```
// ww_mutex 是避免死锁的 mutex
    if (use_ww_ctx) {
        //由 lock 得到 ww_mutex
        ww = container_of(lock, struct ww_mutex, base);
        if (unlikely(ww_ctx == READ_ONCE(ww->ctx)))
            return -EALREADY;
```

```
    }
```

- 尝试获取锁和乐观自旋
  在进行第二步之前，**关闭了内核抢占**。虽然`mutex_lock`在使用中允许进程被调度睡眠，但是在 **上锁** 和 **放锁** 的过程中是不允许被打断的，如果上锁过程中被调度的话，很有可能这个时候被调度的进程正在乐观自旋，进程被调度进入睡眠的时候`osq_lock`锁没有被释放，导致后续进程无法获得相关锁而一直在自旋，无端的消耗CPU资源。

一旦进入下面的循环体，就代表要么成功申请到锁要么成功乐观自旋（基本拿到锁，马上就能结束申请锁的过程了），这个时候就再次开启内核抢占。

```
//尝试获取锁，获取不到就进入 乐观自旋
    if (__mutex_trylock(lock, false) ||
        mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, false)) {
    //进入此模块时，已经成功获取锁
        // lock_acquired 没用的操作
        lock_acquired(&lock->dep_map, ip);
        if (use_ww_ctx)
            ww_mutex_set_context_fastpath(ww, ww_ctx);
        //开启内核抢占
        preempt_enable();
        return 0;
    }
```

- 乐观自旋失败后，进程进入睡眠队列开始等待
  将当前任务加入睡眠队列前需要使用自旋锁将睡眠队列保护起来，随后进入一个循环。在循环中，尝试获取`mutex_lock`锁、判断当前进程是否需要返回RUNNING状态、进行死锁避免的相关维护，执行到这里需要关闭内核抢占（后续可能涉及对等待队列的修改），如果`waiter`是等待队列中的第一个的话，将变量`first`的值改为`true`并将`lock->owner`的`handoff`位变为1。随后再进行一次乐观自旋和拿取锁的尝试，如果还是失败就重新再来一次循环。

```
    for (;;) {

        //成果获取锁，跳出
        //乐观自旋失败就是因为循环时被调度了
        //也有可能调度过程中锁又回到当前任务上
        if (__mutex_trylock(lock, first))
            goto acquired;

        //signal_pending_state 函数判定进程是否需要立即返回 RUNNING 状态
        if (unlikely(signal_pending_state(state, task))) {
            ret = -EINTR; //ret 倒数第三位置 1
            goto err;
        }

        if (use_ww_ctx && ww_ctx->acquired > 0) {
            ret = __ww_mutex_lock_check_stamp(lock, ww_ctx);
            if (ret)
                goto err;
        }

        spin_unlock_mutex(&lock->wait_lock, flags);
    //关闭内核抢占
        schedule_preempt_disabled();
    //将 lock 的 handoff 位置1后，释放锁时需要匹配 FLAGS
        //如果 waiter 是等待队列的第一个就将 first 改为 true
        if (!first && __mutex_waiter_is_first(lock, &wa1iter)) {
            first = true;
            //将 lock 的 handoff位 置为1
            __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
        }
```

```
        //设置任务状态
        set_task_state(task, state);

        //如果是睡眠队列第一个且正在乐观自旋的话，或者尝试获取锁成功，跳出循环
        if ((first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, true)) ||
             __mutex_trylock(lock, first))
            break;

        spin_lock_mutex(&lock->wait_lock, flags);
    }
```

这里对函数 `__mutex_waiter_is_first` 做一下补充：

```
static inline bool __mutex_waiter_is_first(struct mutex *lock, struct mutex_waiter *waiter)
{
    return list_first_entry(&lock->wait_list, struct mutex_waiter, list) == waiter;
}
```

这些函数间的调用关系如下：

```
list_first_entry -> list_entry ->container_of
```

对于函数 `container_of` 有：

```
#ifndef container_of
#define container_of(ptr, type, member) ({          \
    const typeof(((type *)0)->member) * __mptr = (ptr); \
    (type *)((char *)__mptr - offsetof(type, member)); })
#endif
```

函数 `container_of` 的作用就是 **通过结构体的某个成员变量来获得这个结构体的起始位置**。设计如下demo:

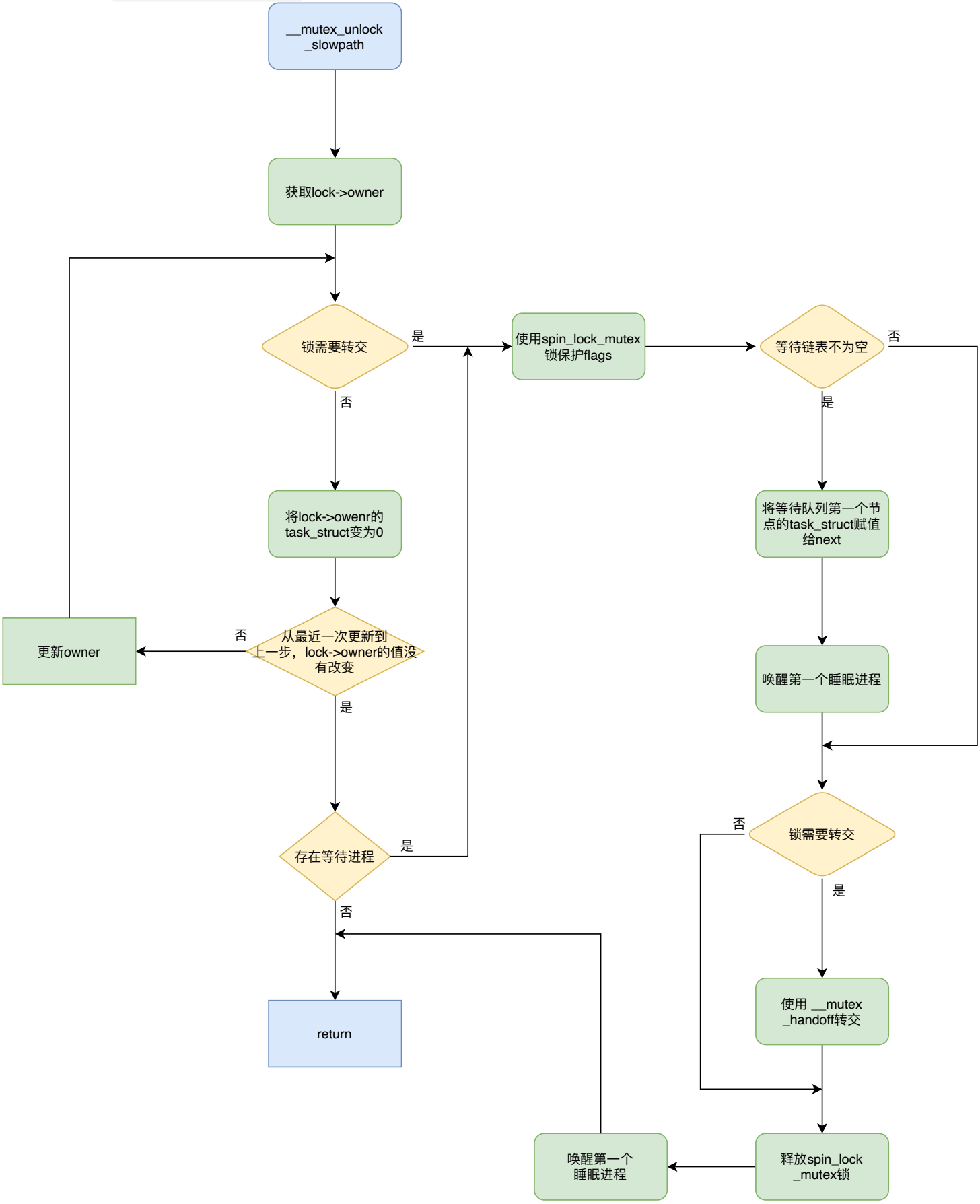
```
#include<stdio.h>

struct test
{
    char i ;
    int j;
    char k;
};

int main()
{
    struct test temp;
    printf("&temp = %p\n",&temp);
    printf("&temp.k = %p\n",&temp.k);
    printf("&((struct test *)0)->k = %d\n",((int)&((struct test *)0)->k));

}
```

运行程序，控制台输出如下：

```
&temp = 0xbf9815b4
&temp.k = 0xbf9815bc
&((struct test *)0)->k = 8
```

C语言结构体存在内存对齐问题，整个结构体的大小是12字节， `&((struct test *)0)->k` 的作用就是求 k 到结构体 `temp` 的起始地址有有几个字节。将0进行强制类型转换后就代表指向该结构体起始位置的指针。因此函数 `container_of` 的作用就是通过结构体的成员变量得到结构体的起始位置。

现在来看函数 `__mutex_lock_common` 的执行逻辑：



函数的总体结构如下：

```
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)
```

```
{   //获取当前任务
    struct task_struct *task = current;
    struct mutex_waiter waiter;
    unsigned long flags;
    bool first = false;
    struct ww_mutex *ww;
    int ret;


//尝试获取锁，获取不到就进入 乐观自旋
    if (__mutex_trylock(lock, false) ||
        mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, false)) {
    //进入此模块时，已经成功获取锁
        ...
        return 0;
    }
//从这里开始，是乐观自旋失败，进程进入睡眠
    //使用自旋锁对睡眠链表进行保护
    spin_lock_mutex(&lock->wait_lock, flags);

    //再次尝试获取锁，成功获取后跳过等待
    if (__mutex_trylock(lock, false))
        goto skip_wait;

//尝试获取锁失败开始进入等待

    //将获取锁失败的任务加入睡眠等待队列的末尾
    list_add_tail(&waiter.list, &lock->wait_list);
    //如果waiter是睡眠等待队列的第一个就将 lock->owner 的最后一位置为 1
    //waiter在睡眠队列里面排第一就表示之前睡眠队列为空，


    for (;;) {

        //成功获取锁，跳出
        if (__mutex_trylock(lock, first))
            goto acquired;

        //signal_pending_state 函数判定进程是否需要立即返回 RUNNING 状态
        if (unlikely(signal_pending_state(state, task))) {
            ...
        }


        //如果是睡眠队列第一个且正在乐观自旋的话，或者尝试获取锁成功，跳出循环
        if ((first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, true)) ||
             __mutex_trylock(lock, first))
            break;

    }


acquired:
    ...

skip_wait:
    ...
    return 0;

err:
    ...
    return ret;
```

```
}
```

掌握了整个函数的框架就可以看详细实现了：

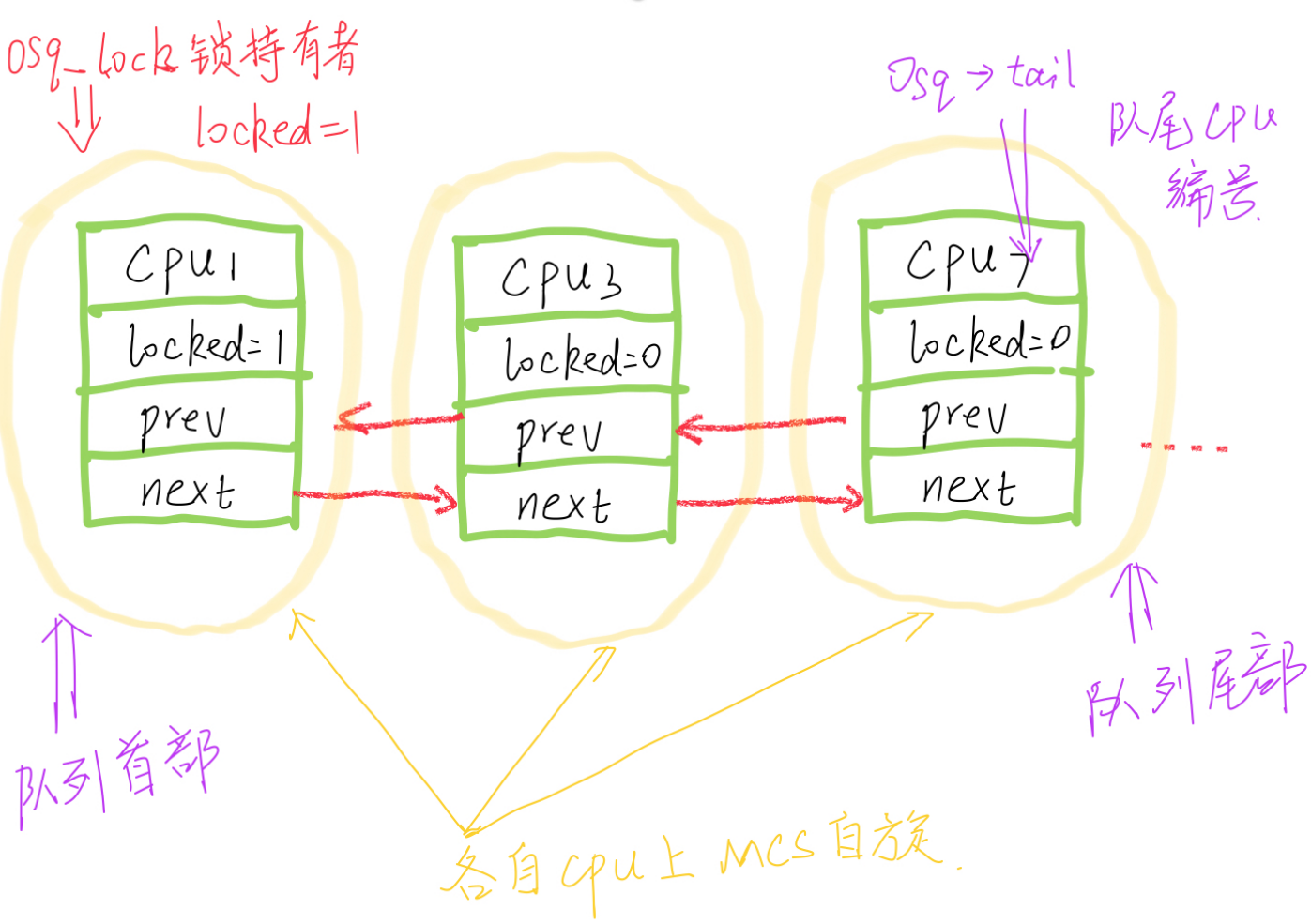
```
static __always_inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass,
            struct lockdep_map *nest_lock, unsigned long ip,
            struct ww_acquire_ctx *ww_ctx, const bool use_ww_ctx)
{   //获取当前任务
    struct task_struct *task = current;
    struct mutex_waiter waiter;
    unsigned long flags;
    bool first = false;
    struct ww_mutex *ww;
    int ret;

// ww_mutex 是避免死锁的 mutex
    if (use_ww_ctx) {
        //由 lock 得到 ww_mutex
        ww = container_of(lock, struct ww_mutex, base);
        if (unlikely(ww_ctx == READ_ONCE(ww->ctx)))
            return -EALREADY;
    }

    //关闭内核抢占
    preempt_disable();
    //没用的函数
    mutex_acquire_nest(&lock->dep_map, subclass, 0, nest_lock, ip);

//尝试获取锁，获取不到就进入 乐观自旋
    if (__mutex_trylock(lock, false) ||
        mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, false)) {
    //进入此模块时，已经成功获取锁
        // lock_acquired 没用的操作
        lock_acquired(&lock->dep_map, ip);
        if (use_ww_ctx)
            ww_mutex_set_context_fastpath(ww, ww_ctx);
        //开启内核抢占
        preempt_enable();
        return 0;
    }
//从这里开始，是乐观自旋失败，进程进入睡眠
    //使用自旋锁对睡眠链表进行保护
    spin_lock_mutex(&lock->wait_lock, flags);

    //再次尝试获取锁，成功获取后跳过等待
    if (__mutex_trylock(lock, false))
        goto skip_wait;

//尝试获取锁失败开始进入等待

    debug_mutex_lock_common(lock, &waiter);
    debug_mutex_add_waiter(lock, &waiter, task);


    //将获取锁失败的任务加入睡眠等待队列的末尾
    list_add_tail(&waiter.list, &lock->wait_list);
    waiter.task = task;
    //如果waiter是睡眠等待队列的第一个就将 lock->owner 的最后一位置为 1
    //waiter在睡眠队列里面排第一就表示之前睡眠队列为空，
    //所以需要将lock->owner最后一位的 waiter 变为0
    if (__mutex_waiter_is_first(lock, &waiter))
```

```
        __mutex_set_flag(lock, MUTEX_FLAG_WAITERS);
    //lock_contended 没用的操作
    lock_contended(&lock->dep_map, ip);

    set_task_state(task, state);
    for (;;) {

        //成果获取锁，跳出
        //乐观自旋失败就是因为循环时被调度了
        //也有可能调度过程中锁又回到当前任务上
        if (__mutex_trylock(lock, first))
            goto acquired;

                //signal_pending_state 函数判定进程是否需要立即返回 RUNNING 状态
        if (unlikely(signal_pending_state(state, task))) {
            ret = -EINTR; //ret 倒数第三位置 1
            goto err;
        }

        if (use_ww_ctx && ww_ctx->acquired > 0) {
            ret = __ww_mutex_lock_check_stamp(lock, ww_ctx);
            if (ret)
                goto err;
        }

        spin_unlock_mutex(&lock->wait_lock, flags);
    //关闭内核抢占
        schedule_preempt_disabled();
    //将 lock 的 handoff 位置1后，释放锁时需要匹配 FLAGS
        //如果 waiter 是等待队列的第一个就将 first 改为 true
        if (!first && __mutex_waiter_is_first(lock, &wa1iter)) {
            first = true;
            //将 lock 的 handoff位 置为1
            __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
        }
        //设置任务状态
        set_task_state(task, state);

        //如果是睡眠队列第一个且正在乐观自旋的话，或者尝试获取锁成功，跳出循环
        if ((first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, true)) ||
             __mutex_trylock(lock, first))
            break;

        spin_lock_mutex(&lock->wait_lock, flags);
    }
    spin_lock_mutex(&lock->wait_lock, flags);
acquired:
    __set_task_state(task, TASK_RUNNING);

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

    debug_mutex_free_waiter(&waiter);

skip_wait:

    //没用的操作
    lock_acquired(&lock->dep_map, ip);

    if (use_ww_ctx)
        ww_mutex_set_context_slowpath(ww, ww_ctx);
```

```
    spin_unlock_mutex(&lock->wait_lock, flags);
    preempt_enable();
    return 0;

err:
    __set_task_state(task, TASK_RUNNING);
    mutex_remove_waiter(lock, &waiter, task);
    spin_unlock_mutex(&lock->wait_lock, flags);
    debug_mutex_free_waiter(&waiter);
    //没用的操作
    mutex_release(&lock->dep_map, 1, ip);
    preempt_enable();
    return ret;
}
```

根据前面的分析，直接获取锁失败之后会尝试进行乐观自旋。在乐观自旋中，首先要获得`osq_lock`锁，当有多个进程申请`osq_lock`锁时，只能有一个进程能拿到锁，其他进程需要循环等待（第一道循环，对应每条快车道尽头的关卡）；拿到了`osq_lock`锁之后就要进入乐观自旋（第二道循环，对应在快车道关口面前等待前面收费站的汽车离开）等待`mutex_lock`的到来。

如下图所示，所有被阻塞在`osq_lock`锁前面的进程都只是在各自的CPU上循环等待（对应的函数在代码实现上循环体内只涉及各自CPU上的变量访问），拥有了`osq_lock`的进程在每一轮循环等待的过程中查看`mutex_lock`的相关变量，尝试获取锁，就是在全局环境下自旋。可以理解为：`osq_lock`将不同CPU上的等待任务串成了一个队列，这个队列的头结点就是`osq_lock`的持有者，头结点在进行乐观自旋，其他结点只是在各自CPU上循环等待。



(4)mutex 解锁实现

`mutex_unlock`释放锁的过程有快速释放和慢速释放，释放锁的进程先尝试快速释放锁，失败后再执行慢速释放锁的过程。

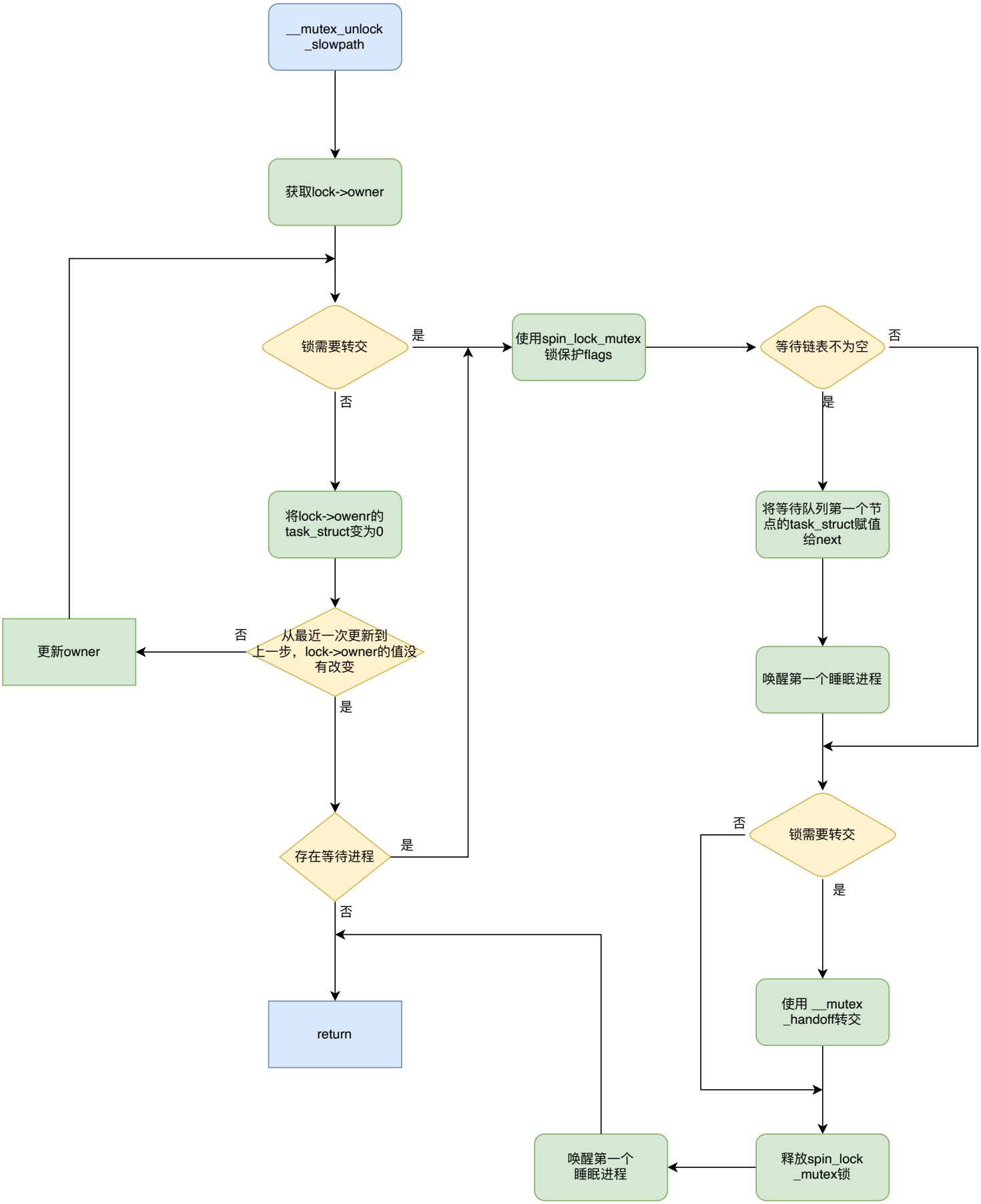
```
//尝试快速解锁失败进入慢速解锁
void __sched mutex_unlock(struct mutex *lock)
{
#ifndef CONFIG_DEBUG_LOCK_ALLOC
    if (__mutex_unlock_fast(lock))
        return;
#endif
    __mutex_unlock_slowpath(lock, _RET_IP_);
}
EXPORT_SYMBOL(mutex_unlock);
```

- 快速释放锁时，如果当前进程是锁持有者，直接将`lock->owner`所有位全部变为0即可。如果更改失败就返回`false`。

```
//快道释放锁
static __always_inline bool __mutex_unlock_fast(struct mutex *lock)
{
    unsigned long curr = (unsigned long)current;
    //如果当前进程是锁持有者，直接释放锁
    //要求 owner 位全为0，除了当前进程是锁持有者 还要 flags 为0
    if (atomic_long_cmpxchg_release(&lock->owner, curr, 0UL) == curr)
        return true;

    return false;
}
```

- 慢速释放锁`__sched __mutex_unlock_slowpath`
  对于慢速释放锁的过程，如果既没有等待进程又不需要转交锁，那么就直接将`mutex->owner`的值变为0。转交锁和存在等待进程的后续处理过程相同：首先判断睡眠等待链表是不是为空，不为空将等待链表的第一个结点信息赋给`lock->owner`信息，锁需要转交的时候修改完`owner`信息后调用函数`__mutex_handoff`,最后唤醒进程，返回函数。

执行逻辑如下：



完整代码实现如下：

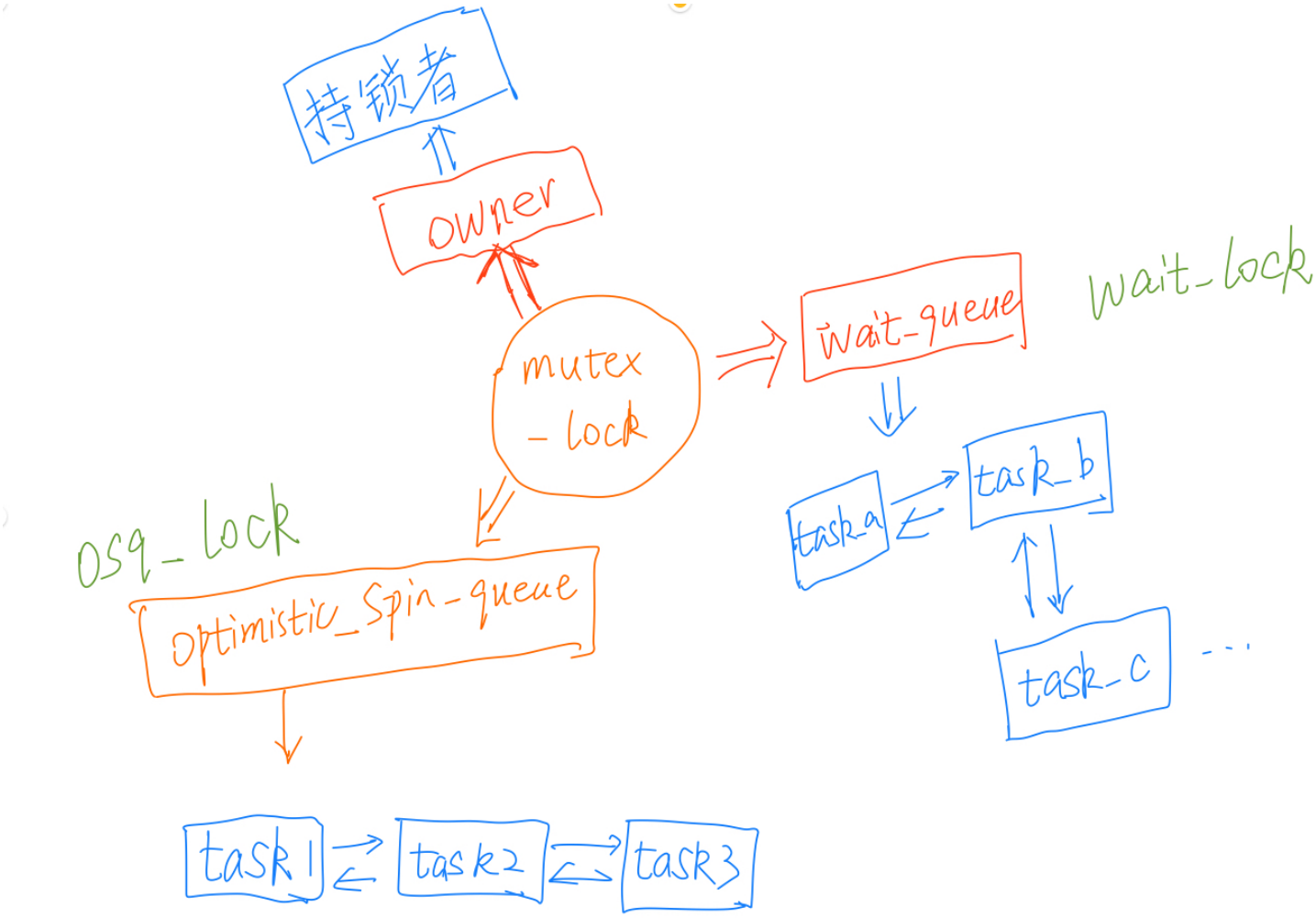
```
static noinline void __sched __mutex_unlock_slowpath(struct mutex *lock, unsigned long ip)
{
    struct task_struct *next = NULL;
    unsigned long owner, flags;
    DEFINE_WAKE_Q(wake_q);
```

```
    //没用的函数
    mutex_release(&lock->dep_map, 1, ip);


    owner = atomic_long_read(&lock->owner);
    //owner handoff 后 wiat_list被置位后退出循环
    for (;;) {
        unsigned long old;

#ifdef CONFIG_DEBUG_MUTEXES
        DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current);
#endif
        //释放锁时需要转交则跳出循环
        if (owner & MUTEX_FLAG_HANDOFF)
            break;
        //到这里当前进程已经放弃了对锁的持有
        old = atomic_long_cmpxchg_release(&lock->owner, owner,
                          __owner_flags(owner));
        //如果有进程阻塞在mutex锁上，跳出循环
        if (old == owner) {
            if (owner & MUTEX_FLAG_WAITERS)
                break;
        //没有进程阻塞，直接返回，释放锁结束
            return;
        }

        //不断更新owner确保owner是最新的值
        owner = old;
    }

    spin_lock_mutex(&lock->wait_lock, flags);
    debug_mutex_unlock(lock);
    if (!list_empty(&lock->wait_list)) {
    //当等待列表不为空时

        //找到等待队列的第一个节点
        struct mutex_waiter *waiter =
            list_first_entry(&lock->wait_list,
                     struct mutex_waiter, list);

        next = waiter->task;
        //没用的函数
        debug_mutex_wake_waiter(lock, waiter);
        //唤醒进程
        wake_q_add(&wake_q, next);
    }
    //如果需要转交，使用handoff函数
    if (owner & MUTEX_FLAG_HANDOFF)
        __mutex_handoff(lock, next);

    spin_unlock_mutex(&lock->wait_lock, flags);
    //唤醒进程
    wake_up_q(&wake_q);
}
```

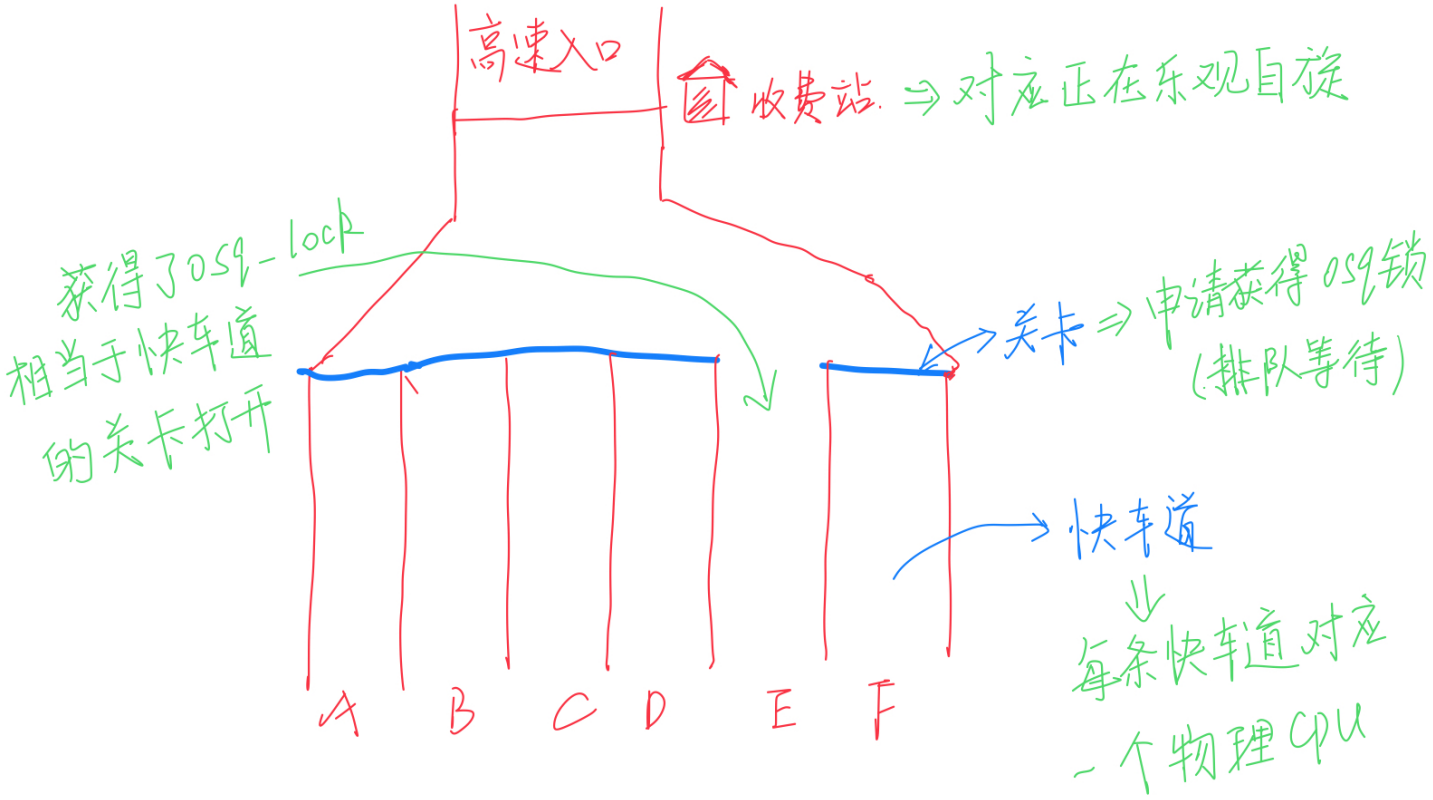
从全局视角了解 Mutex锁

`mutex_lock` 维护了两个队列，一个睡眠等待队列和一个乐观自旋队列还有一个 `unsigned long` 变量 `owner` 用于存储锁持有者信息。乐观自旋队列通过 `osq_lock` 将等待者进行 MCS 自旋，让锁持有者在全局上乐观自旋，乐观自旋失败进入

由`wait_lock`维护的睡眠等待队列，等待被唤醒重新申请锁。



对比前面的场景，可以实现和实列的一一对应：



阅读心得

> 在阅读相关代码的时候一开始感觉无从下手，花了很多时间但是没有进展，后来发现阅读代码不能像阅读文章一样直接往下一读，尤其是这种跟很多文件关联的代码。阅读代码还是要 **由大到小，由粗到细，先了解整体框架，再搞懂细节实现**。了解了整个框架就知道每一块代码大概是干什么的，即使对代码本身读不太懂也可以通过这段代码的目的反过来辅助推断出这段代码的逻辑。
> 有的时候也会出现读懂了代码里面的每一段逻辑但是不知道有什么用这种情况，可以去找一下这段代码被调用的地方，看看是怎么调用的。比如一个函数知道了怎么实现但是不知道有什么用，我们可以通过查找在其他地方对于这段代码的调用，通过 **理解调用这个函数所在的上下文环境** 来推断出这个函数的作用。
> 同时，**结合一些实际场景或者将逻辑可视化** 是一个效率很高的方法。在一些逻辑较为复杂的函数中比如 `mutex_optimistic_spin`，可能一段代码重复读了很多遍过一回还是出现逻辑混乱了可以去画一下对应的流程图，画出图之后感觉整个函数的逻辑清晰了很多，而且后续查看函数逻辑的时候只用直接看图，效率更高。

完整的源码文件如下：

mutex.c

```
/*
 * kernel/locking/mutex.c
 *
 * Mutexes: blocking mutual exclusion locks
 *
 * Started by Ingo Molnar:
 *
 *  Copyright (C) 2004, 2005, 2006 Red Hat, Inc., Ingo Molnar <mingo@redhat.com>
 *
 * Many thanks to Arjan van de Ven, Thomas Gleixner, Steven Rostedt and
 * David Howells for suggestions and improvements.
```

```
 *
 *  - Adaptive spinning for mutexes by Peter Zijlstra. (Ported to mainline
 *    from the -rt tree, where it was originally implemented for rtmutexes
 *    by Steven Rostedt, based on work by Gregory Haskins, Peter Morreale
 *    and Sven Dietrich.
 *
 * Also see Documentation/locking/mutex-design.txt.
 */
#include <linux/mutex.h>
#include <linux/ww_mutex.h>
#include <linux/sched.h>
#include <linux/sched/rt.h>
#include <linux/export.h>
#include <linux/spinlock.h>
#include <linux/interrupt.h>
#include <linux/debug_locks.h>
#include <linux/osq_lock.h>

#ifdef CONFIG_DEBUG_MUTEXES
# include "mutex-debug.h"
#else
# include "mutex.h"
#endif

void
__mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key)
{
    //将owner初始化为0，同时初始化 自旋锁 wait_lock 和链表 wait_list
    atomic_long_set(&lock->owner, 0);
    spin_lock_init(&lock->wait_lock);
    //wait_lock 的作用在于保护等待队列
    INIT_LIST_HEAD(&lock->wait_list);
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    osq_lock_init(&lock->osq);
#endif

    debug_mutex_init(lock, name, key);
}
EXPORT_SYMBOL(__mutex_init);

/*
 * @owner: contains: 'struct task_struct *' to the current lock owner,
 * NULL means not owned. Since task_struct pointers are aligned at
 * ARCH_MIN_TASKALIGN (which is at least sizeof(void *)), we have low
 * bits to store extra state.
 *
 * Bit0 indicates a non-empty waiter list; unlock must issue a wakeup.
 * Bit1 indicates unlock needs to hand the lock to the top-waiter
 */
#define MUTEX_FLAG_WAITERS  0x01 //owner的最后一位 为1时表示有任务在阻塞等待
#define MUTEX_FLAG_HANDOFF  0x02 //owner的倒数第二位
    //在前面的进程释放锁之后会将锁转交给等待队列的第一个进程，置1时表示转交

#define MUTEX_FLAGS     0x03

//获取当前锁持有者信息    owner的前30位
static inline struct task_struct *__owner_task(unsigned long owner)
{
    return (struct task_struct *)(owner & ~MUTEX_FLAGS);
}

//获取 owner 后两位
```

```
static inline unsigned long __owner_flags(unsigned long owner)
{
    return owner & MUTEX_FLAGS;
}

/*
 * Actual trylock that will work on any unlocked state.
 *
 * When setting the owner field, we must preserve the low flag bits.
 *
 * Be careful with @handoff, only set that in a wait-loop (where you set
 * HANDOFF) to avoid recursive lock attempts.
 */

//尝试直接获取锁
static inline bool __mutex_trylock(struct mutex *lock, const bool handoff)
{
    //获取当前锁持有者状态
    unsigned long owner, curr = (unsigned long)current;
    //提取出 lock->owner
    owner = atomic_long_read(&lock->owner);
    for (;;) { /* must loop, can race against a flag */
            //将owner的最后两位提取出来给 flag
        unsigned long old, flags = __owner_flags(owner);
            //如果 owner 的前30位存在,
    // __owner_task 函数返回值不为空，即owner前30位不为0时
        //进入下面的代码块
        if (__owner_task(owner)) {
            //当锁需要转交 且 owner 的前30位与 current 位相同表示获取锁成功
                //owner前30位与current相同代表锁持有者是当前进程
            if (handoff && unlikely(__owner_task(owner) == current)) {
                /*
                 * Provide ACQUIRE semantics for the lock-handoff.
                 *
                 * We cannot easily use load-acquire here, since
                 * the actual load is a failed cmpxchg, which
                 * doesn't imply any barriers.
                 *
                 * Also, this is a fairly unlikely scenario, and
                 * this contains the cost.
                 */
            //此时已经成功获得锁，使用内存屏障让所有竞争者知道锁已经转交
                smp_mb(); /* ACQUIRE */
                return true;
            }
            //当不需要转交或锁被其他任务持有，返回false
            return false;
        }

//注意：代码能运行到这里，肯定是没有进入上面的 IF 判断体的
//也就是锁还没有被持有
        /*
         * We set the HANDOFF bit, we must make sure it doesn't live
         * past the point where we acquire it. This would be possible
         * if we (accidentally) set the bit on an unlocked mutex.
         */

//下面的代码目的是，在被唤醒线程通过转交获取锁的时候，需要将owner的转交位重新置0
        //如果锁需要转交，将flags的倒数第二位置为0
        if (handoff)
            flags &= ~MUTEX_FLAG_HANDOFF;
```

```
        //利用原子操作再次确认这段时间内锁没有被其他人拿走
        //转交锁之前，修改锁的 flag 标识位
        old = atomic_long_cmpxchg_acquire(&lock->owner, owner, curr | flags);
        if (old == owner)
            return true;

        owner = old;
    }
}

#ifndef CONFIG_DEBUG_LOCK_ALLOC
/*
 * Lockdep annotations are contained to the slow paths for simplicity.
 * There is nothing that would stop spreading the lockdep annotations outwards
 * except more code.
 */

/*
 * Optimistic trylock that only works in the uncontended case. Make sure to
 * follow with a __mutex_trylock() before failing.
 */

//直接加锁
static __always_inline bool __mutex_trylock_fast(struct mutex *lock)
{
    unsigned long curr = (unsigned long)current;
    //如果 lock->owner 全为0，代表锁没有持有者，可以直接获取
    if (!atomic_long_cmpxchg_acquire(&lock->owner, 0UL, curr))
        return true;

    return false;
}

//快道释放锁
static __always_inline bool __mutex_unlock_fast(struct mutex *lock)
{
    unsigned long curr = (unsigned long)current;
    //如果当前进程是锁持有者，直接释放锁
    //要求 owner 位全为0，除了当前进程是锁持有者 还要 flags 为0
    if (atomic_long_cmpxchg_release(&lock->owner, curr, 0UL) == curr)
        return true;

    return false;
}
#endif

static inline void __mutex_set_flag(struct mutex *lock, unsigned long flag)
{
    //使用 原子的 或 操作进行置位
    atomic_long_or(flag, &lock->owner);
}

static inline void __mutex_clear_flag(struct mutex *lock, unsigned long flag)
{
    //使用 原子的 与非 操作进行置位  实现flags位清零
    atomic_long_andnot(flag, &lock->owner);
}

static inline bool __mutex_waiter_is_first(struct mutex *lock, struct mutex_waiter *waiter)
{
    return list_first_entry(&lock->wait_list, struct mutex_waiter, list) == waiter;
}
```

```
/*
 * Give up ownership to a specific task, when @task = NULL, this is equivalent
 * to a regular unlock. Clears HANDOFF, preserves WAITERS. Provides RELEASE
 * semantics like a regular unlock, the __mutex_trylock() provides matching
 * ACQUIRE semantics for the handoff.
 */

//将锁转交给 task
static void __mutex_handoff(struct mutex *lock, struct task_struct *task)
{
    unsigned long owner = atomic_long_read(&lock->owner);

    for (;;) {
        unsigned long old, new;

#ifdef CONFIG_DEBUG_MUTEXES
        DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current);
#endif
        //将 owner 最后一位赋值给 new 的最后一位
        new = (owner & MUTEX_FLAG_WAITERS);
        //将 new 前30位 置为 task 的值
        new |= (unsigned long)task;
        //锁没有被其他进程拿走的话，将 new 赋值给 lock->owner，完成锁的转交
        old = atomic_long_cmpxchg_release(&lock->owner, owner, new);
        //赋值成功的话，退出循环
        if (old == owner)
            break;

        owner = old;
    }
}

#ifndef CONFIG_DEBUG_LOCK_ALLOC
/*
 * We split the mutex lock/unlock logic into separate fastpath and
 * slowpath functions, to reduce the register pressure on the fastpath.
 * We also put the fastpath first in the kernel image, to make sure the
 * branch is predicted by the CPU as default-untaken.
 */
static void __sched __mutex_lock_slowpath(struct mutex *lock);

/**
 * mutex_lock - acquire the mutex
 * @lock: the mutex to be acquired
 *
 * Lock the mutex exclusively for this task. If the mutex is not
 * available right now, it will sleep until it can get it.
 *
 * The mutex must later on be released by the same task that
 * acquired it. Recursive locking is not allowed. The task
 * may not exit without first unlocking the mutex. Also, kernel
 * memory where the mutex resides must not be freed with
 * the mutex still locked. The mutex must first be initialized
 * (or statically defined) before it can be locked. memset()-ing
 * the mutex to 0 is not allowed.
 *
 * ( The CONFIG_DEBUG_MUTEXES .config option turns on debugging
 *   checks that will enforce the restrictions and will also do
 *   deadlock debugging. )
 *
 * This function is similar to (but not equivalent to) down().
```

```
 */

//锁的获取
void __sched mutex_lock(struct mutex *lock)
{
    //可能会睡眠
    might_sleep();

    //尝试获取快道失败就获取慢道
//这里的逻辑从 mutex锁 整体来看，先尝试直接获取锁，
            //拿不到的话就尝试自旋等待、睡眠等待
    if (!__mutex_trylock_fast(lock))
        __mutex_lock_slowpath(lock);
}
EXPORT_SYMBOL(mutex_lock);
#endif

static __always_inline void ww_mutex_lock_acquired(struct ww_mutex *ww,
                            struct ww_acquire_ctx *ww_ctx)
{
#ifdef CONFIG_DEBUG_MUTEXES
    /*
     * If this WARN_ON triggers, you used ww_mutex_lock to acquire,
     * but released with a normal mutex_unlock in this call.
     *
     * This should never happen, always use ww_mutex_unlock.
     */
    DEBUG_LOCKS_WARN_ON(ww->ctx);

    /*
     * Not quite done after calling ww_acquire_done() ?
     */
    DEBUG_LOCKS_WARN_ON(ww_ctx->done_acquire);

    if (ww_ctx->contending_lock) {
        /*
         * After -EDEADLK you tried to
         * acquire a different ww_mutex? Bad!
         */
        DEBUG_LOCKS_WARN_ON(ww_ctx->contending_lock != ww);

        /*
         * You called ww_mutex_lock after receiving -EDEADLK,
         * but 'forgot' to unlock everything else first?
         */
        DEBUG_LOCKS_WARN_ON(ww_ctx->acquired > 0);
        ww_ctx->contending_lock = NULL;
    }

    /*
     * Naughty, using a different class will lead to undefined behavior!
     */
    DEBUG_LOCKS_WARN_ON(ww_ctx->ww_class != ww->ww_class);
#endif
    ww_ctx->acquired++;
}

/*
 * After acquiring lock with fastpath or when we lost out in contested
 * slowpath, set ctx and wake up any waiters so they can recheck.
 */
static __always_inline void
```

```
ww_mutex_set_context_fastpath(struct ww_mutex *lock,
                  struct ww_acquire_ctx *ctx)
{
    unsigned long flags;
    struct mutex_waiter *cur;

    ww_mutex_lock_acquired(lock, ctx);

    lock->ctx = ctx;

    /*
     * The lock->ctx update should be visible on all cores before
     * the atomic read is done, otherwise contended waiters might be
     * missed. The contended waiters will either see ww_ctx == NULL
     * and keep spinning, or it will acquire wait_lock, add itself
     * to waiter list and sleep.
     */
    smp_mb(); /* ^^^ */

    /*
     * Check if lock is contended, if not there is nobody to wake up
     */
    if (likely(!(atomic_long_read(&lock->base.owner) & MUTEX_FLAG_WAITERS)))
        return;

    /*
     * Uh oh, we raced in fastpath, wake up everyone in this case,
     * so they can see the new lock->ctx.
     */
    spin_lock_mutex(&lock->base.wait_lock, flags);
    list_for_each_entry(cur, &lock->base.wait_list, list) {
        debug_mutex_wake_waiter(&lock->base, cur);
        wake_up_process(cur->task);
    }
    spin_unlock_mutex(&lock->base.wait_lock, flags);
}

/*
 * After acquiring lock in the slowpath set ctx and wake up any
 * waiters so they can recheck.
 *
 * Callers must hold the mutex wait_lock.
 */
static __always_inline void
ww_mutex_set_context_slowpath(struct ww_mutex *lock,
                  struct ww_acquire_ctx *ctx)
{
    struct mutex_waiter *cur;

    ww_mutex_lock_acquired(lock, ctx);
    lock->ctx = ctx;

    /*
     * Give any possible sleeping processes the chance to wake up,
     * so they can recheck if they have to back off.
     */
    list_for_each_entry(cur, &lock->base.wait_list, list) {
        debug_mutex_wake_waiter(&lock->base, cur);
        wake_up_process(cur->task);
    }
}
```

```c
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
/*
 * Look out! "owner" is an entirely speculative pointer
 * access and not reliable.
 */
static noinline
//锁 被其他 owner 持有时，直接返回true
//被owner持有时，不断循环，等待 owner 被抢占调度，返回 false（该等待过程被加锁了）
//从 mutex锁 的机制来看，当mutex被其他进程持有时，直接返回true；
    //锁被本进程持有时，不断循环检测当前进程是否被调度
//只有在owner持有锁的时候放好被调度了才户返回false，不然都是true
bool mutex_spin_on_owner(struct mutex *lock, struct task_struct *owner)
{
    bool ret = true;
    //获得rcu读锁，进入临界区
    rcu_read_lock();
    // __mutex_owner 的作用是通过 lock 获得锁的持有者
    //如果 mutex 锁被 owner 持有，不断自旋等待
    //除非持有者被调度，返回 false
    while (__mutex_owner(lock) == owner) {
        /*
         * Ensure we emit the owner->on_cpu, dereference _after_
         * checking lock->owner still matches owner. If that fails,
         * owner might point to freed memory. If it still matches,
         * the rcu_read_lock() ensures the memory stays valid.
         */
        barrier();

        /*
         * Use vcpu_is_preempted to detect lock holder preemption issue.
         */
        //当锁持有者进程在睡眠、被调度、内核被抢占时返回 false
        if (!owner->on_cpu || need_resched() ||
                vcpu_is_preempted(task_cpu(owner))) {
            ret = false;
            break;
        }

        cpu_relax();
    }
    rcu_read_unlock();

    return ret;
}

/*
 * Initial check for entering the mutex spinning loop
 */
// mutex 锁被持有且持有者正在执行，不允许抢占、调度时返回 1，否则返回 0
//返回1时，lock->owner可以自旋，返回0就不可以自旋
static inline int mutex_can_spin_on_owner(struct mutex *lock)
{
    //lock->owner 指向该进程的 task_struct结构
    // task_struct->on cpu 为1时表示锁持有者正在临界区运行，
    //当锁被锁持有者释放后， lock->owner 为 NULL
    struct task_struct *owner;
    int retval = 1;
    //内核被调度，返回0
    if (need_resched())
        return 0;

//使用rcu_lock来创建临界区是为了 owner 指针所指的 task_struct
```

```
  // 不会因为你进程被杀而导致访问owner指针出错
  //RCU指针可以保护 task_struct 结构在临界区不被释放
    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)
    //当前锁持有者正在执行且关闭了内核抢占时，retval的值为1，否则为0
        retval = owner->on_cpu && !vcpu_is_preempted(task_cpu(owner));
    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.
     */
    //如果返回0，说明当前锁持有者并不在临界区执行，或内核可以被抢占，或锁空闲
    return retval;
}

/*
 * Optimistic spinning.
 *
 * We try to spin for acquisition when we find that the lock owner
 * is currently running on a (different) CPU and while we don't
 * need to reschedule. The rationale is that if the lock owner is
 * running, it is likely to release the lock soon.
 *
 * The mutex spinners are queued up using MCS lock so that only one
 * spinner can compete for the mutex. However, if mutex spinning isn't
 * going to happen, there is no point in going through the lock/unlock
 * overhead.
 *
 * Returns true when the lock was taken, otherwise false, indicating
 * that we need to jump to the slowpath and sleep.
 *
 * The waiter flag is set to true if the spinner is a waiter in the wait
 * queue. The waiter-spinner will spin on the lock directly and concurrently
 * with the spinner at the head of the OSQ, if present, until the owner is
 * changed to itself.
 */

//乐观自旋锁
//该函数返回 true 时，代表结束自旋且拿到mutex锁
static bool mutex_optimistic_spin(struct mutex *lock,
                  struct ww_acquire_ctx *ww_ctx,
                  const bool use_ww_ctx, const bool waiter)
{
    //当前进程的 task_struct 结构
    struct task_struct *task = current;
    //当前进程不是等待进程时
    //注意：这里的 waiter 与owner的最后一位代表的 waiter不同
    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().
```

```
         */
        //mutex锁 拥有者在睡眠或被调度时，不能自旋等待
        if (!mutex_can_spin_on_owner(lock))
            goto fail;

        /*
         * 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.
         */
        //获取 osq 锁失败，跳转 fail
        if (!osq_lock(&lock->osq))
            goto fail;
    }
//如果等待队列不为空，或者等待队列为空时锁拥有者正在运行且无法被打断且获得 osq 锁
    //进入下面的循环，也就是乐观自旋，在乐观自旋中不断循环尝试三件事
    for (;;) {
        struct task_struct *owner;
//乐观自旋做的第一件事：
    //避免发生死锁
        if (use_ww_ctx && ww_ctx->acquired > 0) {
            struct ww_mutex *ww;

            //由 lock 得到 struct ww_mutex
            ww = container_of(lock, struct ww_mutex, base);
            /*
             * If ww->ctx is set the contents are undefined, only
             * by acquiring wait_lock there is a guarantee that
             * they are not invalid when reading.
             *
             * As such, when deadlock detection needs to be
             * performed the optimistic spinning cannot be done.
             */
            // ww 存在上下文就跳转到 fail_unlock
            if (READ_ONCE(ww->ctx))
                goto fail_unlock;
        }

        /*
         * If there's an owner, wait for it to either
         * release the lock or go to sleep.
         */
//乐观自旋做的第二件事：
    //判断锁被持有的情况下；
    //1、判断持有者是否是当前进程且有任务在等待队列中
    //2、判断锁持有者是否被调度进入睡眠
        //获取锁持有者信息
        owner = __mutex_owner(lock);
        if (owner) {
            //有任务正在等待且当前任务持有锁 退出乐观自旋
            //这里 owner==task 判断成功的意思应该是mutex锁的前一个持有者
                //将锁给转交过来的 。
            //可以理解为是前一个持有者将 owner 设置为 task 的
            if (waiter && owner == task) {
                smp_mb(); /* ACQUIRE */
                break;
            }
        //到这里要么是没有等待任务，要么是锁持有者不是当前任务
            //当锁被owner持有且持有者最终被调度时 跳转到 fail_unlock
            if (!mutex_spin_on_owner(lock, owner))
                //进入这个 if 结构只可能是owner是lock的持有者且被调度了
                //锁持有者被调度进入睡眠，无法解锁
```

```
                goto fail_unlock;
        }

        /* Try to acquire the mutex if it is unlocked. */
        //当 waiter 成功拿到锁时跳出循环（也是退出乐观自旋）
//乐观自旋做的第三件事：
    //尝试获取 mutex锁
        if (__mutex_trylock(lock, waiter))
            break;

        /*
         * The cpu_relax() call is a compiler barrier which forces
         * everything in this loop to be re-loaded. We don't need
         * memory barriers as we'll eventually observe the right
         * values at the cost of a few extra spins.
         */
        cpu_relax();
    }
//如果代码能够运行到这里，代表着 owner 现在已经是 mutex锁 的持有者
    //如果没有等待任务，释放 osq锁
    if (!waiter)
        osq_unlock(&lock->osq);
//乐观自旋函数一旦返回true，代表已经结束自旋并且获得锁了
    return true;


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

fail:
    /*
     * If we fell out of the spin path because of need_resched(),
     * reschedule now, before we try-lock the mutex. This avoids getting
     * scheduled out right after we obtained the mutex.
     */
    //如果其他任务抢占CPU
    if (need_resched()) {
        /*
         * We _should_ have TASK_RUNNING here, but just in case
         * we do not, make it so, otherwise we might get stuck.
         */
        //保存当前状态
        __set_current_state(TASK_RUNNING);
        //禁止内核抢占
        schedule_preempt_disabled();
    }

    return false;
}
#else
static bool mutex_optimistic_spin(struct mutex *lock,
                  struct ww_acquire_ctx *ww_ctx,
                  const bool use_ww_ctx, const bool waiter)
{
    return false;
}
#endif

static noinline void __sched __mutex_unlock_slowpath(struct mutex *lock, unsigned long ip);

/**
```

```
 * mutex_unlock - release the mutex
 * @lock: the mutex to be released
 *
 * Unlock a mutex that has been locked by this task previously.
 *
 * This function must not be used in interrupt context. Unlocking
 * of a not locked mutex is not allowed.
 *
 * This function is similar to (but not equivalent to) up().
 */
//尝试快速解锁失败进入慢速解锁
void __sched mutex_unlock(struct mutex *lock)
{
#ifndef CONFIG_DEBUG_LOCK_ALLOC
    if (__mutex_unlock_fast(lock))
        return;
#endif
    __mutex_unlock_slowpath(lock, _RET_IP_);
}
EXPORT_SYMBOL(mutex_unlock);

/**
 * ww_mutex_unlock - release the w/w mutex
 * @lock: the mutex to be released
 *
 * Unlock a mutex that has been locked by this task previously with any of the
 * ww_mutex_lock* functions (with or without an acquire context). It is
 * forbidden to release the locks after releasing the acquire context.
 *
 * This function must not be used in interrupt context. Unlocking
 * of a unlocked mutex is not allowed.
 */

void __sched ww_mutex_unlock(struct ww_mutex *lock)
{
    /*
     * The unlocking fastpath is the 0->1 transition from 'locked'
     * into 'unlocked' state:
     */
    if (lock->ctx) {
#ifdef CONFIG_DEBUG_MUTEXES
        DEBUG_LOCKS_WARN_ON(!lock->ctx->acquired);
#endif
        if (lock->ctx->acquired > 0)
            lock->ctx->acquired--;
        lock->ctx = NULL;
    }

    mutex_unlock(&lock->base);
}
EXPORT_SYMBOL(ww_mutex_unlock);

static inline int __sched
__ww_mutex_lock_check_stamp(struct mutex *lock, struct ww_acquire_ctx *ctx)
{
    struct ww_mutex *ww = container_of(lock, struct ww_mutex, base);
    struct ww_acquire_ctx *hold_ctx = READ_ONCE(ww->ctx);

    if (!hold_ctx)
        return 0;

    if (ctx->stamp - hold_ctx->stamp <= LONG_MAX &&
```

```
        (ctx->stamp != hold_ctx->stamp || ctx > hold_ctx)) {
#ifdef CONFIG_DEBUG_MUTEXES
        DEBUG_LOCKS_WARN_ON(ctx->contending_lock);
        ctx->contending_lock = ww;
#endif
        return -EDEADLK;
    }

    return 0;
}

/*
 * Lock a mutex (possibly interruptible), slowpath:
 */
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)
{   //获取当前任务
    struct task_struct *task = current;
    struct mutex_waiter waiter;
    unsigned long flags;
    bool first = false;
    struct ww_mutex *ww;
    int ret;

// ww_mutex 是避免死锁的 mutex
    if (use_ww_ctx) {
        //由 lock 得到 ww_mutex
        ww = container_of(lock, struct ww_mutex, base);
        if (unlikely(ww_ctx == READ_ONCE(ww->ctx)))
            return -EALREADY;
    }

    //关闭内核抢占
    preempt_disable();
    //没用的函数
    mutex_acquire_nest(&lock->dep_map, subclass, 0, nest_lock, ip);

//尝试获取锁，获取不到就进入 乐观自旋
    if (__mutex_trylock(lock, false) ||
        mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, false)) {
    //进入此模块时，已经成功获取锁
        // lock_acquired 没用的操作
        lock_acquired(&lock->dep_map, ip);
        if (use_ww_ctx)
            ww_mutex_set_context_fastpath(ww, ww_ctx);
        //开启内核抢占
        preempt_enable();
        return 0;
    }
//从这里开始，是乐观自旋失败，进程进入睡眠
    //使用自旋锁对睡眠链表进行保护
    spin_lock_mutex(&lock->wait_lock, flags);

    //再次尝试获取锁，成功获取后跳过等待
    if (__mutex_trylock(lock, false))
        goto skip_wait;

//尝试获取锁失败开始进入等待

    debug_mutex_lock_common(lock, &waiter);
```

```
    debug_mutex_add_waiter(lock, &waiter, task);

    /* add waiting tasks to the end of the waitqueue (FIFO): */
    //将获取锁失败的任务加入睡眠等待队列的末尾
    list_add_tail(&waiter.list, &lock->wait_list);
    waiter.task = task;
    //如果waiter是睡眠等待队列的第一个就将 lock->owner 的最后一位置为 1
    //waiter在睡眠队列里面排第一就表示之前睡眠队列为空,
    //所以需要将lock->owner最后一位的 waiter 变为0
    if (__mutex_waiter_is_first(lock, &waiter))
        __mutex_set_flag(lock, MUTEX_FLAG_WAITERS);
    //lock_contended 没用的操作
    lock_contended(&lock->dep_map, ip);

    set_task_state(task, state);
    for (;;) {
        /*
          Once we hold wait_lock, we're serialized against
         mutex_unlock() handing the lock off to us, do a trylock
          before testing the error conditions to make sure we pick up
          the handoff.
         */
        //成果获取锁，跳出
        //乐观自旋失败就是因为循环时被调度了
        //也有可能调度过程中锁又回到当前任务上
        if (__mutex_trylock(lock, first))
            goto acquired;

        /*
          Check for signals and wound conditions while holding
          wait_lock. This ensures the lock cancellation is ordered
          against mutex_unlock() and wake-ups do not go missing.
         */

        //signal_pending_state 函数判定进程是否需要立即返回 RUNNING 状态
        if (unlikely(signal_pending_state(state, task))) {
            ret = -EINTR; //ret 倒数第三位置 1
            goto err;
        }

        if (use_ww_ctx && ww_ctx->acquired > 0) {
            ret = __ww_mutex_lock_check_stamp(lock, ww_ctx);
            if (ret)
                goto err;
        }

        spin_unlock_mutex(&lock->wait_lock, flags);
    //关闭内核抢占
        schedule_preempt_disabled();
    //将 lock 的 handoff 位置1后，释放锁时需要匹配 FLAGS
        //如果 waiter 是等待队列的第一个就将 first 改为 true
        if (!first && __mutex_waiter_is_first(lock, &wa1iter)) {
            first = true;
            //将 lock 的 handoff位 置为1
            __mutex_set_flag(lock, MUTEX_FLAG_HANDOFF);
        }
        //设置任务状态
        set_task_state(task, state);
        /*
         * Here we order against unlock; we must either see it change
         * state back to RUNNING and fall through the next schedule(),
         * or we must see its unlock and acquire.
```

```
         */
        //如果是睡眠队列第一个且正在乐观自旋的话，或者尝试获取锁成功，跳出循环
        if ((first && mutex_optimistic_spin(lock, ww_ctx, use_ww_ctx, true)) ||
             __mutex_trylock(lock, first))
            break;

        spin_lock_mutex(&lock->wait_lock, flags);
    }
    spin_lock_mutex(&lock->wait_lock, flags);
acquired:
    __set_task_state(task, TASK_RUNNING);

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

    debug_mutex_free_waiter(&waiter);

skip_wait:
    /* got the lock - cleanup and rejoice! */
    //没用的操作
    lock_acquired(&lock->dep_map, ip);

    if (use_ww_ctx)
        ww_mutex_set_context_slowpath(ww, ww_ctx);

    spin_unlock_mutex(&lock->wait_lock, flags);
    preempt_enable();
    return 0;

err:
    __set_task_state(task, TASK_RUNNING);
    mutex_remove_waiter(lock, &waiter, task);
    spin_unlock_mutex(&lock->wait_lock, flags);
    debug_mutex_free_waiter(&waiter);
    //没用的操作
    mutex_release(&lock->dep_map, 1, ip);
    preempt_enable();
    return ret;
}

#ifdef CONFIG_DEBUG_LOCK_ALLOC
void __sched
mutex_lock_nested(struct mutex *lock, unsigned int subclass)
{
    might_sleep();
    __mutex_lock_common(lock, TASK_UNINTERRUPTIBLE,
                subclass, NULL, _RET_IP_, NULL, 0);
}

EXPORT_SYMBOL_GPL(mutex_lock_nested);

void __sched
_mutex_lock_nest_lock(struct mutex *lock, struct lockdep_map *nest)
{
    might_sleep();
    __mutex_lock_common(lock, TASK_UNINTERRUPTIBLE,
                0, nest, _RET_IP_, NULL, 0);
}
EXPORT_SYMBOL_GPL(_mutex_lock_nest_lock);

int __sched
```

```
mutex_lock_killable_nested(struct mutex *lock, unsigned int subclass)
{
    might_sleep();
    return __mutex_lock_common(lock, TASK_KILLABLE,
                   subclass, NULL, _RET_IP_, NULL, 0);
}
EXPORT_SYMBOL_GPL(mutex_lock_killable_nested);

int __sched
mutex_lock_interruptible_nested(struct mutex *lock, unsigned int subclass)
{
    might_sleep();
    return __mutex_lock_common(lock, TASK_INTERRUPTIBLE,
                   subclass, NULL, _RET_IP_, NULL, 0);
}
EXPORT_SYMBOL_GPL(mutex_lock_interruptible_nested);

static inline int
ww_mutex_deadlock_injection(struct ww_mutex *lock, struct ww_acquire_ctx *ctx)
{
#ifdef CONFIG_DEBUG_WW_MUTEX_SLOWPATH
    unsigned tmp;

    if (ctx->deadlock_inject_countdown-- == 0) {
        tmp = ctx->deadlock_inject_interval;
        if (tmp > UINT_MAX/4)
            tmp = UINT_MAX;
        else
            tmp = tmp*2 + tmp + tmp/2;

        ctx->deadlock_inject_interval = tmp;
        ctx->deadlock_inject_countdown = tmp;
        ctx->contending_lock = lock;

        ww_mutex_unlock(lock);

        return -EDEADLK;
    }
#endif

    return 0;
}

int __sched
__ww_mutex_lock(struct ww_mutex *lock, struct ww_acquire_ctx *ctx)
{
    int ret;

    might_sleep();
    ret =  __mutex_lock_common(&lock->base, TASK_UNINTERRUPTIBLE,
                   0, &ctx->dep_map, _RET_IP_, ctx, 1);
    if (!ret && ctx->acquired > 1)
        return ww_mutex_deadlock_injection(lock, ctx);

    return ret;
}
EXPORT_SYMBOL_GPL(__ww_mutex_lock);

int __sched
__ww_mutex_lock_interruptible(struct ww_mutex *lock, struct ww_acquire_ctx *ctx)
{
    int ret;
```

```
    might_sleep();
    ret = __mutex_lock_common(&lock->base, TASK_INTERRUPTIBLE,
                  0, &ctx->dep_map, _RET_IP_, ctx, 1);

    if (!ret && ctx->acquired > 1)
        return ww_mutex_deadlock_injection(lock, ctx);

    return ret;
}
EXPORT_SYMBOL_GPL(__ww_mutex_lock_interruptible);

#endif

/*
 * Release the lock, slowpath:
 */
static noinline void __sched __mutex_unlock_slowpath(struct mutex *lock, unsigned long ip)
{
    struct task_struct *next = NULL;
    unsigned long owner, flags;
    DEFINE_WAKE_Q(wake_q);
    //没用的函数
    mutex_release(&lock->dep_map, 1, ip);

    /*
     * Release the lock before (potentially) taking the spinlock such that
     * other contenders can get on with things ASAP.
     *
     * Except when HANDOFF, in that case we must not clear the owner field,
     * but instead set it to the top waiter.
     */
    owner = atomic_long_read(&lock->owner);
    //owner handoff 后 wiat_list被置位后退出循环
    for (;;) {
        unsigned long old;

#ifdef CONFIG_DEBUG_MUTEXES
        DEBUG_LOCKS_WARN_ON(__owner_task(owner) != current);
#endif
        //释放锁时需要转交则跳出循环
        if (owner & MUTEX_FLAG_HANDOFF)
            break;
        //到这里当前进程已经放弃了对锁的持有
        old = atomic_long_cmpxchg_release(&lock->owner, owner,
                          __owner_flags(owner));
        //如果有进程阻塞在mutex锁上，跳出循环
        if (old == owner) {
            if (owner & MUTEX_FLAG_WAITERS)
                break;
        //没有进程阻塞，直接返回，释放锁结束
            return;
        }

        //不断更新owner确保owner是最新的值
        owner = old;
    }

    spin_lock_mutex(&lock->wait_lock, flags);
    debug_mutex_unlock(lock);
    if (!list_empty(&lock->wait_list)) {
    //当等待列表不为空时
```

```
        /* get the first entry from the wait-list: */
        //找到等待队列的第一个节点
        struct mutex_waiter *waiter =
            list_first_entry(&lock->wait_list,
                     struct mutex_waiter, list);

        next = waiter->task;
        //没用的函数
        debug_mutex_wake_waiter(lock, waiter);
        //唤醒进程
        wake_q_add(&wake_q, next);
    }
    //如果需要转交，使用handoff函数
    if (owner & MUTEX_FLAG_HANDOFF)
        __mutex_handoff(lock, next);

    spin_unlock_mutex(&lock->wait_lock, flags);
    //唤醒进程
    wake_up_q(&wake_q);
}

#ifndef CONFIG_DEBUG_LOCK_ALLOC
/*
 * Here come the less common (and hence less performance-critical) APIs:
 * mutex_lock_interruptible() and mutex_trylock().
 */
static noinline int __sched
__mutex_lock_killable_slowpath(struct mutex *lock);

static noinline int __sched
__mutex_lock_interruptible_slowpath(struct mutex *lock);

/**
 * mutex_lock_interruptible - acquire the mutex, interruptible
 * @lock: the mutex to be acquired
 *
 * Lock the mutex like mutex_lock(), and return 0 if the mutex has
 * been acquired or sleep until the mutex becomes available. If a
 * signal arrives while waiting for the lock then this function
 * returns -EINTR.
 *
 * This function is similar to (but not equivalent to) down_interruptible().
 */
int __sched mutex_lock_interruptible(struct mutex *lock)
{
    might_sleep();

    if (__mutex_trylock_fast(lock))
        return 0;

    return __mutex_lock_interruptible_slowpath(lock);
}

EXPORT_SYMBOL(mutex_lock_interruptible);

int __sched mutex_lock_killable(struct mutex *lock)
{
    might_sleep();

    if (__mutex_trylock_fast(lock))
        return 0;
```

```
    return __mutex_lock_killable_slowpath(lock);
}
EXPORT_SYMBOL(mutex_lock_killable);

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

static noinline int __sched
__mutex_lock_killable_slowpath(struct mutex *lock)
{
    return __mutex_lock_common(lock, TASK_KILLABLE, 0,
                   NULL, _RET_IP_, NULL, 0);
}

static noinline int __sched
__mutex_lock_interruptible_slowpath(struct mutex *lock)
{
    return __mutex_lock_common(lock, TASK_INTERRUPTIBLE, 0,
                   NULL, _RET_IP_, NULL, 0);
}

static noinline int __sched
__ww_mutex_lock_slowpath(struct ww_mutex *lock, struct ww_acquire_ctx *ctx)
{
    return __mutex_lock_common(&lock->base, TASK_UNINTERRUPTIBLE, 0,
                   NULL, _RET_IP_, ctx, 1);
}

static noinline int __sched
__ww_mutex_lock_interruptible_slowpath(struct ww_mutex *lock,
                       struct ww_acquire_ctx *ctx)
{
    return __mutex_lock_common(&lock->base, TASK_INTERRUPTIBLE, 0,
                   NULL, _RET_IP_, ctx, 1);
}

#endif

/**
 * mutex_trylock - try to acquire the mutex, without waiting
 * @lock: the mutex to be acquired
 *
 * Try to acquire the mutex atomically. Returns 1 if the mutex
 * has been acquired successfully, and 0 on contention.
 *
 * NOTE: this function follows the spin_trylock() convention, so
 * it is negated from the down_trylock() return values! Be careful
 * about this when converting semaphore users to mutexes.
 *
 * This function must not be used in interrupt context. The
 * mutex must be released by the same task that acquired it.
 */
int __sched mutex_trylock(struct mutex *lock)
{
    bool locked = __mutex_trylock(lock, false);

    if (locked)
    // mutex_acquire 没用
```

```
        mutex_acquire(&lock->dep_map, 0, 1, _RET_IP_);

    return locked;
}
EXPORT_SYMBOL(mutex_trylock);

#ifndef CONFIG_DEBUG_LOCK_ALLOC
int __sched
__ww_mutex_lock(struct ww_mutex *lock, struct ww_acquire_ctx *ctx)
{
    might_sleep();

    if (__mutex_trylock_fast(&lock->base)) {
        ww_mutex_set_context_fastpath(lock, ctx);
        return 0;
    }

    return __ww_mutex_lock_slowpath(lock, ctx);
}
EXPORT_SYMBOL(__ww_mutex_lock);

int __sched
__ww_mutex_lock_interruptible(struct ww_mutex *lock, struct ww_acquire_ctx *ctx)
{
    might_sleep();

    if (__mutex_trylock_fast(&lock->base)) {
        ww_mutex_set_context_fastpath(lock, ctx);
        return 0;
    }

    return __ww_mutex_lock_interruptible_slowpath(lock, ctx);
}
EXPORT_SYMBOL(__ww_mutex_lock_interruptible);

#endif

/**
 * atomic_dec_and_mutex_lock - return holding mutex if we dec to 0
 * @cnt: the atomic which we are to dec
 * @lock: the mutex to return holding if we dec to 0
 *
 * return true and hold lock if we dec to 0, return false otherwise
 */
int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock)
{
    /* dec if we can't possibly hit 0 */
    if (atomic_add_unless(cnt, -1, 1))
        return 0;
    /* we might hit 0, so take the lock */
    mutex_lock(lock);
    if (!atomic_dec_and_test(cnt)) {
        /* when we actually did the dec, we didn't hit 0 */
        mutex_unlock(lock);
        return 0;
    }
    /* we hit 0, and we hold the lock */
    return 1;
}
EXPORT_SYMBOL(atomic_dec_and_mutex_lock);
```

osq_lock.c

```
#include <linux/percpu.h>
#include <linux/sched.h>
#include <linux/osq_lock.h>


static DEFINE_PER_CPU_SHARED_ALIGNED(struct optimistic_spin_node, osq_node);

//由CPU编号获得在 osq_lock 中对应的CPU编号
static inline int encode_cpu(int cpu_nr)
{
    return cpu_nr + 1;
}

//由 MCS锁中的 optimistic_spin_queue 结构获得系统内的CPU编号
static inline int node_cpu(struct optimistic_spin_node *node)
{
    return node->cpu - 1;
}

//将 函数 encode_cpu 编码出来的CPU编号解码成系统对应的CPU编号
static inline struct optimistic_spin_node *decode_cpu(int encoded_cpu_val)
{
    int cpu_nr = encoded_cpu_val - 1;

    return per_cpu_ptr(&osq_node, cpu_nr);
}

//该函数的作用是获取 osq 队列中的下一个节点
static inline struct optimistic_spin_node *
osq_wait_next(struct optimistic_spin_queue *lock,
          struct optimistic_spin_node *node,
          struct optimistic_spin_node *prev)
{
    struct optimistic_spin_node *next = NULL;
    int curr = encode_cpu(smp_processor_id());
    int old;


    //如果prev 的 cpu 编号存在，将其赋值给 old。否则将 0 赋给 old
    old = prev ? prev->cpu : OSQ_UNLOCKED_VAL;

     //lock->tail==curr 代表当前节点是链表的最后一个节点，
     //以下循环主要是为了实现：
     //1、当前节点为最后一个节点时，删除该节点，返回NULL
     //2、当前节点存在后继节点时，解除前驱节点对当前节点的指针
     //   找到当前节点的后继节点并返回
    for (;;) {
        //如果 lock->tail 为当前进程cpu 编号，
        //将前驱节点的 cpu编号赋给 lock->tail,跳出循环

        if (atomic_read(&lock->tail) == curr &&
            atomic_cmpxchg_acquire(&lock->tail, curr, old) == curr) {

            break;
        }
        //如果当前节点存在后继节点，将 node->next 赋值为null，跳出循环
        if (node->next) {
            next = xchg(&node->next, NULL);
            if (next)
```

```
                break;
        }

        cpu_relax();
    }
    return next;
}

bool osq_lock(struct optimistic_spin_queue *lock)
{
    //当前节点的CPU编号
    struct optimistic_spin_node *node = this_cpu_ptr(&osq_node);

    struct optimistic_spin_node *prev, *next;
    //将处理当前进程的CPU编号进行编码
    int curr = encode_cpu(smp_processor_id());
    int old;
    //将 locked 赋值为0表示当前节点并未持有锁
    node->locked = 0;
    node->next = NULL;
    node->cpu = curr;



    //使用原子操作将 当前CPU编码结果赋值给 lock 中的原子变量 tail ，
      // 并返回赋值前 lock->tail 的值，将其赋给 old
    old = atomic_xcchg(&lock->tail, curr);

    //如果赋值前 lock->tail 的值为0，那么这时当前进程可以直接获得锁，即申请锁成功
    //这里的意思是如果 lock->tail 的值为0，那么代表锁的前一个持有者已经执行完毕
    if (old == OSQ_UNLOCKED_VAL)
        return true;
//以下为赋值前 lock->tail 的值不为0的情况
//lock->tail不为0代表锁的持有者还在占用CPU运行
    //执行链表的节点插入操作（将当前节点插入链表）
    prev = decode_cpu(old);
    node->prev = prev;
    WRITE_ONCE(prev->next, node);



    //不停循环，直到当前节点获得锁
    while (!READ_ONCE(node->locked)) {


        //如果进程被调度或睡眠，跳出当前循环，进入 unqueue
        if (need_resched() || vcpu_is_preempted(node_cpu(node->prev)))
            goto unqueue;

        cpu_relax();
    }
    return true;

unqueue:


    for (;;) {

        //如果当前节点和其前驱节点的关系没有改变 将当前节点脱离链表并跳出循环
        if (prev->next == node &&
            cmpxchg(&prev->next, node, NULL) == node)
            break;
```

```
        //如果此时获得了锁就返回 true
        //前面已经判断过一次了为什么还要判断一下此时有没有拿到锁呢?
        //因为这里 node 的前继节点已经被改变则证明前一时刻锁被别的进程拿走了,
        //这里需要再判断一下是不是刚好别的进程退出临界区并释放了锁, 被当前进程拿到
        if (smp_load_acquire(&node->locked))
            return true;

        cpu_relax();


        //程序运行到这里 prev 已经不是 node 的前驱节点, 需要重新获取
        prev = READ_ONCE(node->prev);
    }


    //由 osq_wait_next 代码可知, 返回的 next 是 node 的后继节点
    next = osq_wait_next(lock, node, prev);
    //如果 next 为空, 代表被删除时 node 已经是最后一个节点,
       //直接返回 false,没拿到锁
    if (!next)
        return false;




    //如果 next 不为空, 将将继节点的 next 指向后继节点,
      //后继节点的 prev 指向前继节点。 返回false
    WRITE_ONCE(next->prev, prev);
    WRITE_ONCE(prev->next, next);

    return false;
}

void osq_unlock(struct optimistic_spin_queue *lock)
{
    struct optimistic_spin_node *node, *next;
    int curr = encode_cpu(smp_processor_id());

    //如果当前节点是最后一个节点, 直接将 lock->tail 设置为0 表示锁空闲, 返回函数
    if (likely(atomic_cmpxchg_release(&lock->tail, curr,
                      OSQ_UNLOCKED_VAL) == curr))
        return;

    //如果当前节点不是最后一个节点, 通过原子操作获取后继节点
    //将后继节点 locked 设为 1, 表示持有锁
    node = this_cpu_ptr(&osq_node);
    next = xchg(&node->next, NULL);
    if (next) {
        WRITE_ONCE(next->locked, 1);
        return;
    }
    //获取后继节点失败, 等待下一个后继节点出现
    next = osq_wait_next(lock, node, NULL);
    if (next)
        WRITE_ONCE(next->locked, 1);
}
```