方法	描述
mutex_lock(struct mutex*)	为指定的mutex上锁，如果不可用则睡眠
mutex_unlock(struct mutex*)	为指定的mutex解锁
mutex_trylock(struct mutex*)	视图获取指定的mutex，如果成功则返回1；否则锁被获取，返回值是0
mutex_is_lock(struct mutex*)	如果锁已被征用，则返回1；否则返回0

mutex的简洁性和高效性源自于相比使用信号量更多的受限性。它不同于信号量，因为mutex仅仅实现了Dijkstra设计初衷中的最基本的行为。因此mutex的使用场景相对而言更严格。

(1)代码：linux/kernel/mutex.c

```
void inline fastcall __sched mutex_lock(struct mutex *lock);
    //??????
```

实际上是先给count做自减操作，然后使用本身的自旋锁进入临界区操作。首先取得count的值，在将count置为 - 1，判断如果原来count的置为1，也即互斥锁可以获得，则直接获取，跳出。否则进入循环反复测试互斥锁的状态。在循环中，也是先取得互斥锁原来的状态，在将其之为 - 1，判断如果可以获取(等于1)，则退出循环，否则设置当前进程的状态为不可中断状态，解锁自身的自旋锁，进入睡眠状态，待被在调度唤醒时，再获得自身的自旋锁，进入新一次的查询其自身状态(该互斥锁的状态)的循环。

(2)具体参见linux/kernel/mutex.c

```
int fastcall __sched mutex_lock_interruptible(struct mutex *
lock)?
```

和mutex_lock()一样，也是获取互斥锁。在获得了互斥锁或进入睡眠直到获得互斥锁之后会返回0。如果在等待获取锁的时候进入睡眠状态收到一个信号(被信号打断睡眠)，则返回_EINIR。

(3)具体参见linux/kernel/mutex.c

```
int fastcall __sched mutex_trylock(struct mutex *lock);
```

试图获取互斥锁，如果成功获取则返回1，否则返回0，不等待。

释放互斥锁

具体参见linux/kernel/mutex.c

```
void fastcall mutex_unlock(struct mutex *lock);
```

释放被当前进程获取的互斥锁。该函数不能用在中断上下文中，而且不允许去释放一个没有上锁的互斥锁。

互斥锁试用注意事项

- 任何时刻中只有一个任务可以持有mutex,
  也就是说，mutex的使用计数永远是1
- 给mutex锁者必须负责给其再解锁——你不能在一个上下文中锁定一个mutex，而在另一个上下文中给它解锁。这个限制使得mutex不适合内核同用户空间复杂的同步场景。最
  常使用的方式是：在同一上下文中上锁和解锁。
- 递归地上锁和解锁是不允许的。也就是说，你不能递归地持有同一个锁，同样你也不能再去解锁一个已经被解开的mutex
- 当持有一个mutex时 ，进程不可以退出
- mutex不能在中断或者下半部中使用，即使使用mutex_trylock()也不行
- mutex只能通过官方API管理：它只能使用上节中描述的方法初始化，不可被拷贝、手动 初始化或者重复初始化

信号量和互斥体

互斥体和信号量很相似，内核中两者共存会令人混淆。所幸，它们的标准使用方式

都有简单规范：除非mutex的某个约束妨碍你使用，否则相比信号量要优先使用mutex。当你写新代码时，只有碰到特殊场合（一般是很底层代码）才会需要使用信号量。因此建议选mutex。如果发现不能满足其约束条件，且没有其他别的选择时，再考虑选择信号量

自旋锁和互斥体使用场合

了解何时使用自旋锁，何时使用互斥体（或信号量）对编写优良代码很重要，但是多数情况下，并不需要太多的考虑，因为在中断上下文中只能使用自旋锁，而在任务睡眠时只能使用互斥体。

下面总结一下各种锁的需求情况