Linux 内核模块的强制删除-结束 rmmod 这类 disk sleep 进程

一.问题：

在测试Hello驱动时发现 ,当rmmod一个模块的时候 ,在模块的exit函数中阻塞了 ,rmmod 进程杀也杀不掉，永远呆在那里，发现它已经是 D(disk sleep)状态了.

二.分析：

既然是在内核中出了问题 ,还是需要在内核中寻找办法 ,解决这类问题的前提是对内核卸载模块的精确理解. 我按照这个原则，查到了 rmmod 最终调用的代码：

```
asmlinkage long sys_delete_module(const char __user *name_user,
unsigned int flags)
{
    ...

    if (!list_empty(&mod->modules_which_use_me)) { //0.如果其它模块
依赖该模块，则不删除
        /* Other modules depend on us: get rid of them first. */
        ret = -EWOULDBLOCK;
        goto out;
    }
    ...

    if (mod->state != MODULE_STATE_LIVE) { //1.如果模块不是 LIVE 状态，
那么就无法进行下去了，得到的结果是 busy
        ...
        ret = -EBUSY;
        goto out;
    }
    ...

    if (!forced && module_refcount(mod) != 0) //2.如果引用计数不是 0，
则等待其为 0
        wait_for_zero_refcount(mod);
    up(&module_mutex);

    mod->exit();     //3.如果在这个里面阻塞了，那就无法返回了
```

```
    down(&module_mutex);
    free_module(mod);
    ...
}
```

以上注释了 4 处，分别解释如下：

情况 0： 有其它模块依赖要卸载的模块。模块 a 是否依赖模块 b，这个在模块 a 加载的时候调用 resolve_symbol 抉择，如果模块 a 的 symbol 在模块 b 中，则依赖

情况 1： 只有 LIVE 状态的模块才能被卸载。

情况 2： 引用计数在有其它模块或者内核本身引用的时候不为 0，要卸载就要等待它们不引用为止。

情况 3： 这个情况比较普遍，因为模块万一在使用过程中 oom 或者依赖它的模块 oom 或者模块本身写的不好有 bug 从而破坏了一些数据结构，那么可能造成 exit 函数中阻塞，最终 rmmod 不返回！

三.尝试一下：

针对情况 3，举一个例子来模拟：

```
static DECLARE_COMPLETION(test_completion);
int init_module()
{
       return 0;
}
void cleanup_module( )
{
    wait_for_completion(&test_completion);
}
MODULE_LICENSE("GPL");
```

编译为 test.ko，最终在 rmmod test 的时候会阻塞，rmmod 永不返回了！很显然是 cleanup_module 出了问题，因此再写一个模块来卸载它！在编译模块之前，首先要在 /proc/kallsym 中找到以下这行：

f88de380 d __this_module [XXXX 无法卸载的模块名称]

这是因为 modules 链表没有被导出，如果被导出的话，正确的方式应该是遍历这个链表来比对模块名称的。

以下的模块加载了以后，上述模块就可以被 rmmod 了：

```
void force(void)
{
}
int __init rm_init(void)
{
	struct module *mod = (struct module*)0xf88de380;
	int i;
	int o=0;

	mod->state = MODULE_STATE_LIVE; //为了卸载能进行下去，也就是避开情况 1，将模块的状态改变为 LIVE

	mod->exit = force;	//由于是模块的 exit 导致了无法返回，则替换 mod 的 exit。再次调用 rmmod 的时候会调用到 sys_delete_module，最后会调用 exit 回调函数，新的 exit 当然不会阻塞，成功返回，进而可以 free 掉 module

	for (i = 0; i < NR_CPUS; i++) { //将引用计数归 0
		mod->ref[i].count = *(local_t *)&o;
	}
	return 0;
}
void __exit rm_exit(void)
{
}
module_init(rm_init);
module_exit(rm_exit);
MODULE_LICENSE("GPL");
```

然后加载上述模块后，前面的模块就可以卸载了。

四.更深入的一些解释：

针对这个模块导致的无法卸载的问题，还有另一种方解决式，就是在别的 module 中

complete 掉这个 completion，这个 completion 当然是无法直接得到的，还是要通过 /proc/kallsyms 得到这个 completion 的地址，然后强制转换成 completion 并完成它：

```
int __init rm_init(void)
{
       complete((struct completion *)0xf88de36c);
       return 0;
}
void __exit rm_exit(void)
{
}
module_init(rm_init);
module_exit(rm_exit);
MODULE_LICENSE("GPL");
```

当然这种方式是最好的了，否则如果使用替换 exit 的方式，会导致前面的那个阻塞的 rmmod 无法退出。你可能想在新编写的模块中调用 try_to_wake_up 函数，然而这也是不妥当的，因为它可能在 wait_for_completion，而 wait_for_completion 中大量引用了已经被替换 exit 回调函数进而被卸载的模块数据，比如：

```
spin_unlock_irq(&x->wait.lock);
schedule();
spin_lock_irq(&x->wait.lock);
```

其中 x 就是那个模块里面的 ,既然 x 已经没有了(使用替换 exit 的方式已经成功卸载了模块，模块被 free ,x 当然就不复存在了) ,刚刚被唤醒运行的 rmmod 就会 oops ,但是不管怎样，一个进程的 oops 一般是没有问题的，因此还是可以干掉它的，这种 oops 一般不会破坏其它的内核数据，一般都是由于引用已经被 free 的指针引起的(当然还真的可能有危险情况哦...)。 既然知道这些 rmmod 都是阻塞在睡眠里面，那么我们只需要强制唤醒它们就可以了 ,至于说被唤醒后 oops 了怎么办？由内核处理啦 ,或者听天由命 !因此考虑以下的代码：

```
int (*try)(task_t * p, unsigned int state, int sync);
int __init rm_init(void){
       struct task_struct *tsk = find_task_by_pid(28792); //28792
为一个阻塞的 rmmod 进程 ,这个模块使用上述的替换 exit 的方式已经被重新 rmmod 卸
```

```
载了，然而第一次的那个 rmmod 仍然阻塞在哪里，没有睡再去唤醒它了。
        try=0xc011a460;
        (*try)(tsk, TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE, 0); //
我们唤醒它，至于它醒了之后干什么，随便吧！
        return 0;
}
void __exit rm_exit(void){
}
module_init(rm_init);
module_exit(rm_exit);
MODULE_LICENSE("GPL");
```

然后再 ps -e 一下，基本没有那个 rmmod 进程了。一个[State: D (disk sleep)]的进程这样完蛋了。

以上代码基本都是硬编码的地址以及进程号，真正的代码应该使用参数来传递这些信息，就会比较方便了！

既然模块结构都可以拿到，它的任意字段就可以被任意赋值，哪里出了问题就重新赋值哪里！既然内核空间都进入了，导不导出符号就不是根本问题了，就算没有 procfs 的 kallsym，也一样能搞定，因为你能控制整个内存！

五.防删除：

我们可以在自己的模块初始化的时候将其引用计数设置成一个比较大的数 ,或者设置一下别的模块结构体字段，防治被 rmmod，然而别人也可以再写一个模块把这些字段设置回去，简单的使用上述方式就可以干掉你的防删除模块，这是一个矛与盾的问题，关键是，别让人拥有 root 权限。

六.总结：

解决模块由于阻塞而无法删除问题有下面的过程：

1.写一个模块替换 exit 函数，且设置引用计数为 0，状态为 LIVE，然后 rmmod；

2.强制 try_to_wake_up 那个 rmmod 进程，注意不能使用 wake_up，因为队列可能已经

不在了，而应该直接唤醒 task_struct；

附：内核缺页

在 do_page_fault 中，如果缺页发生在内核空间，最终 OOPS 的话，会调用 die：

die("Oops", regs, error_code);

在 die 中，如果没有处在中断以及没有设置 panic-on-oops 的话，最终将以 SIGSEGV 退出当前进程：

```
if (in_interrupt())
    panic("Fatal exception in interrupt");
if (panic_on_oops) {
    printk(KERN_EMERG "Fatal exception: panic in 5 seconds\n");
    set_cu rrent_state(TASK_UNINTERRUPTIBLE);
    schedule_timeout(5 * HZ);
    panic("Fatal exception");
}
do_exit(SIGSEGV);
```

这样，如果唤醒睡眠在模块 exit 中的 rmmod，显然在被唤醒之后，检测变量会导致缺页(由于变量已经被 free 了)，因此会进入 die("Oops"...)，最终退出 rmmod 进程，这个也是很合理的哦！因此上述的清理 D 状态的进程还是可以用的。