

2024 年全国大学生计算机系统能力大赛-操作系统设计赛(全国)-OS 功能挑战赛道

比赛题目 proj62 Linux 锁优先级传递

队 伍 名 我爱说实话

队 伍 成 员 曲晨阳

卫策

王淑华

时 间 2024 年 5 月 27 日

一、 比赛准备和调研

1. 题目描述：

优先级翻转是指当一个高优先级任务通过锁等机制访问共享资源时，该锁如果已被一低优先级任务占有，将可能造成高优先级任务被许多具有较低优先级任务阻塞。优先级翻转问题是影响 linux 实时性的障碍之一，甚至引起很多稳定性问题。

该项目的目标：

- 实现 rwlock、rwsem、mutex 的优先级继承支持
- 通过内核宏开关优先级继承相关特性

2. Linux 内核自旋锁（spinlock）实现逻辑

自旋锁是指当一个线程在获取锁时，如果锁已经被其它 CPU 中的线程获取，那么该线程将循环等待，并不断判断是否能够成功获取锁，直到其它 CPU 释放锁后，等待锁的线程才会退出循环。

自旋锁只能被一个任务持有，由于其他请求自旋锁的 CPU 在循环等待锁，因此为了防止死锁，持有自旋锁的 CPU 不允许进入睡眠，不允许上下文切换，禁止调度。

在 Linux 内核中，自旋锁是一个无符号整型值的 lock 变量，自旋锁结构体和函数关键逻辑需要依赖于体系结构的实现。spinlock 的核心思想是基于 tickets 的机制，每个锁的数据结构 arch_spinlock_t 中维护两个字段：next 和 owner，只有当 next 和 owner 相等时才能获取锁；每个进程在获取锁的时候，next 值会增加，当进程在释放锁的时候 owner 值会增加；如果有多个进程在争抢锁的时候，类似一个排队系统，FIFO ticket spinlock。

3. 读写锁（rwlock）和读写信号量（rwsem）实现逻辑

读写锁（rwlock）基于自旋锁实现，又称读写自旋锁，读写锁的机制是一种多读单写的锁机制，这种自旋锁为读和写分别提供了不同的锁。一个或多个读任务可以并发地持有读任务锁；相反，用于写的锁最多只能被一个写任务持有，而且此时不能有并发的读操作。有时把读/写锁叫做共享/排斥锁，或者并发/排斥锁，因为这种锁以共享（对读任务而言）和排斥（对写任务而言）的形式获得使用。

读写信号量（rwsem）与读写锁类似，但是基于信号量实现，读写信号量同样分为读信号量和写信号量，读任务在持有读写信号量期间只能对该读写信号量保护的共享资源进行读访问，如果一个任务除了需要读，可能还需要写，那么它必须被归类为写任务，它在对共享资源访问之前必须先获得写任务身份，写任务在发现自己不需要写访问的情况下可以降级为读任务。读写信号量同时拥有的读任务数不受限制，也就是说可以有任意多个读任务同时拥有一个读写信号量。如果一个读写信号量当前没有被读任务或写任务拥有并且也没有写任务等待该信号量，那么一个写任务可以成功获得该读写信号量，否则写任务将被挂起，直到没有任何访问者。因此写任务是排他性的，独占性的。

两者的区别是，读写信号量适合于保护更长的临界区，以防止并行访问。不适合用于保护较短的代码范围，因为竞争信号量时需要使进程睡眠和再次唤醒，代价较高。而读写锁适合于保护较短的临界区，

避免 cpu 长期处于忙等状态。

4. 互斥锁（mutex）实现逻辑

互斥锁（mutex）不依赖于信号量，避免了使用信号量所导致的不必要的开销。Linux 内核中互斥量的基本数据结构定义如下：

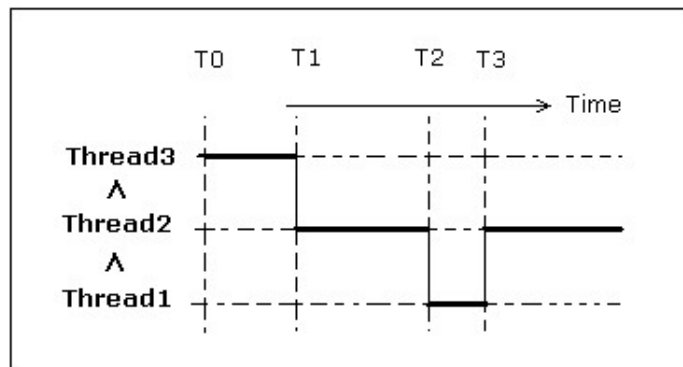
```
struct mutex {  
    /*1:未锁定, 0:锁定, 负值: 锁定, 可能有等待者 */  
    atomic_t count;  
    spinlock_t wait_lock;  
    struct list_head wait_list;  
};
```

如果互斥量未锁定，则 count 为 1。锁定分为两种情况。如果只有一个进程在使用互斥量，则 count 设置为 0。如果互斥量被锁定，而且有进程在等待互斥量解锁（在解锁时需要唤醒等待进程），则 count 为负值。

5. 什么是优先级翻转及解决方法

优先级反转是指低优先级任务先于高优先级任务执行的现象，具体来说就是高优先级任务被低优先级任务阻塞，导致高优先级任务迟迟得不到调度。但其他中等优先级的任务却能抢到 CPU 资源。从运行情况上来看，好像是中优先级的任务比高优先级任务具有更高的优先权。

举一个具体的例子，假定一个进程中有三个线程 Thread1（高）、Thread2（中）和 Thread3（低），考虑下图的执行情况。



其每个时刻的运行过程如下：

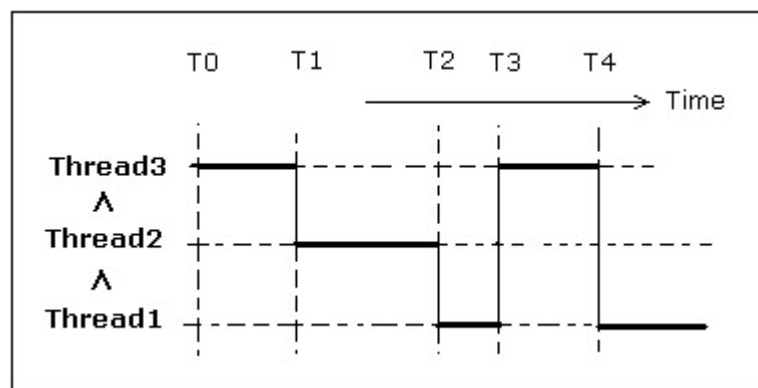
- T0时刻，Thread3 运行，并获得同步资源 SYNCH1；
- T1 时刻，Thread2 开始运行，由于优先级高于 Thread3，Thread3 被抢占（未释放同步资源 SYNCH1），Thread2 被调度执行；
- T2时刻，Thread1 抢占 Thread2；
- T3时刻，Thread1 需要同步资源 SYNCH1，但 SYNCH1 被更低优先级的 Thread3 所拥有，Thread1 被挂起等待该资源

而此时线程 Thread2 和 Thread3 都处于可运行状态，Thread2 的优先级大于 Thread3 的优先级，Thread2 被调度执行。最终的结果是高优先级的 Thread1 迟迟无法得到调度，而中优先级的 Thread2 却能抢到 CPU 资源。

其解决方法有两种：优先级天花板（priority ceiling, PC）和优先级继承（priority inheritance, PI）。

优先级天花板（priority ceiling）是指将申请某资源的任务的优先级提升到可能访问该资源的所有任务中最高优先级任务的优先级。每个临界资源 R 赋予一个优先级天花板，取所有可能访问临界资源 R 的所有进程中优先级最高的作为 R 的优先级天花板。当一个进程获取到临界资源，就将该进程的优先级提升到优先级天花板，这样，该进程不会被其他可能使用该临界资源的进程抢占，从而得到更快执行，更快地释放临界资源。释放临界资源后，恢复优先级。

优先级继承（priority inheritance）是指当高优先级进程（ $P1$ ）请求一个已经被低优先级（ $P3$ ）占有的临界资源时，将低优先级进程（ $P3$ ）的优先级临时提升到与高优先级进程一样的级别，使得低优先级进程能更快地运行，从而更快地释放临界资源。低优先级进程离开临界区后，其优先级恢复至原本的值。回到上述例子，拥有优先级继承的线程调度如下图所示。



到了 $T3$ 时刻，Thread1 需要 Thread3 占用的同步资源 SYNCH1，操作系统检测到这种情况后，就把 Thread3 的优先级提高到 Thread1 的优先级。此时处于可运行状态的线程 Thread2 和 Thread3 中，Thread3 的优先级大于 Thread2 的优先级，Thread3 被调度执行。

Thread3 执行到 $T4$ 时刻，释放了同步资源 SYNCH1，操作系统恢复了 Thread3 的优先级，Thread1 获得了同步资源 SYNCH1，重新进入可执行队列。处于可运行状态的线程 Thread1 和 Thread2 中，Thread1 的优先级大于 Thread2 的优先级，所以 Thread1 被调度执行。

二、 算法设计原理

本项目旨在实现 rwlock、rwsem、mutex 的优先级继承（PI）支持。如上所述，rwlock 和 rwsem 中的读任务共享临界区，因此本项目直接继承原有的 read_lock 和 read_sem 设计，主要针对 write_lock、write_sem 和 mutex 添加优先级继承支持，其算法的设计思路类似，以下统一介绍其原理。

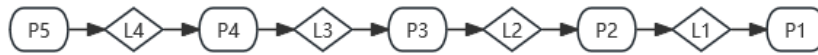
1. 优先级继承链（PI chain）

优先级继承是指锁的低优先级所有者继承高优先级等待者的优先级，直到锁被释放。而在实际问题中，往往不像上述的三个任务的例子那么简单，举一个复杂一点的例子，假设有五个进程： $P1$ 、 $P2$ 、 $P3$ 、 $P4$ 、 $P5$ ，有四个锁： $L1$ 、 $L2$ 、 $L3$ 、 $L4$ ，其运行情况如下：

- $P1$ 拥有锁 $L1$

- P2 拥有锁 L2，等待 L1 释放
- P3 拥有锁 L3，等待 L2 释放
- P4 拥有锁 L4，等待 L3 释放
- P5 等待 L4 释放

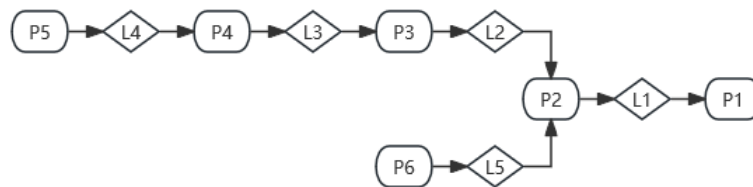
可见多个锁和多个进程相互阻塞，可以抽象为一个阻塞链：



在这条阻塞链中，假如有优先级继承支持，那么 P1 的优先级应来着这条链上的最大优先级，即

$$Prio(P1) = MAX(Prio(P1), Prio(P2), Prio(P3), Prio(P4), Prio(P5))$$

而实际问题中，往往存在多个进程被一个锁阻塞，则阻塞链实际上为树形结构，假如在以上的例子中添加一个新进程 P6 和一个新锁 L5，P6 请求获取 L5，而 L5 被 P2 持有，则 P6 被 P2 阻塞无法运行，阻塞链则更新为：



若此锁拥有优先级继承功能，则要求链的最右侧，也就是链的顶端的优先级始终大于等于左侧的优先级，这就是优先级继承链（PI chain）。

在使用有优先级继承的锁时，始终维护这一个 PI chain，实现优先级的继承与更新。

2. 等待者树

每个锁使用一个红黑树（rbtree）按优先级存储这些等待者。该树受到位于互斥体结构中的自旋锁的保护。这个自旋锁为 wait_lock。

3. 优先级继承树

每个进程使用一个红黑树保存被该进程的锁阻塞的最高优先级等待者，称为优先级继承树（PI rbtree），该树只包含顶部等待者，也就是等待该锁的最高优先级进程，如果进程继承了优先级，那么位于该树顶部的进程的优先级将始终是该进程的优先级。

4. 关键代码描述

本部分使用添加优先级继承的 mutex 的相关代码作为演示，描述算法原理。

pi_mutex 结构体：

```
struct pi_mutex
{
    raw_spinlock_t wait_lock;
    struct rb_root_cached waiters;
    struct task_struct *owner;
```



```

{
    DEFINE_WAKE_Q(wake_q);

    if (likely(pi_mutex_cmpxchg_release(lock, current, NULL)))
        return;

    if (slowfn(lock, &wake_q))
        pi_mutex_postunlock(&wake_q);
}

```

调整任务优先级函数：

`pi_mutex_adjust_prio` 函数检查任务的优先级以及正在等待该任务拥有的任何锁的最高优先级进程。由于任务的 **PI rbtree** 中保存了该任务拥有的所有互斥锁的所有顶级等待者的优先级顺序，因此我们只需将顶级 **pi** 等待者与其自己的正常/截止优先级进行比较，并取较高的优先级。然后调用 `rt_mutex_setprio` 将任务的优先级调整为新的优先级。

// 调整任务优先级

```

static void pi_mutex_adjust_prio(struct task_struct *p)
{
    struct task_struct *pi_task = NULL;

    lockdep_assert_held(&p->pi_lock);

    if (task_has_pi_waiters(p))
        pi_task = task_top_pi_waiter(p)->task;

    pi_mutex_setprio(p, pi_task);
}

```

调整 **PI chain** 函数：

优先级继承功能通过维护一条 **PI chain** 实现，当有锁释放时或者有新任务请求锁的时候都需要对其进行调整

// 调整优先级链

```

static int pi_mutex_adjust_prio_chain(struct task_struct *task,
                                     enum rtmutex_chainwalk chwalk,
                                     struct pi_mutex *orig_lock,
                                     struct pi_mutex *next_lock,
                                     struct pi_mutex_waiter *orig_waiter,
                                     struct task_struct *top_task)
{
    struct pi_mutex_waiter *waiter, *top_waiter = orig_waiter;
    struct pi_mutex_waiter *prerequeue_top_waiter;
    int ret = 0, depth = 0;
    struct pi_mutex *lock;

```

```

    bool detect_deadlock;
    bool requeue = true;
    // 检测死锁
    detect_deadlock = pi_mutex_cond_detect_deadlock(orig_waiter, chwalk);

again:
    if (++depth > max_lock_depth) {
        static int prev_max;

        if (prev_max != max_lock_depth) {
            prev_max = max_lock_depth;
            printk(KERN_WARNING "Maximum lock depth %d reached "
                "task: %s (%d)\n", max_lock_depth,
                top_task->comm, task_pid_nr(top_task));
        }
        put_task_struct(task);

        return -EDEADLK;
    }

retry:
    raw_spin_lock_irq(&task->pi_lock);
    waiter = task->pi_blocked_on;

    // 检查是否到达链尾，或者链的状态在释放锁之后发生了变化。
    if (!waiter)
        goto out_unlock_pi;

    // 检查 orig_waiter 的状态
    if (orig_waiter && !pi_mutex_owner(orig_lock))
        goto out_unlock_pi;

    // 检测 mutex 链是否发生变化
    if (next_lock != waiter->lock)
        goto out_unlock_pi;

    // 如果任务没有等待者，则退出
    if (top_waiter) {
        if (!task_has_pi_waiters(task))
            goto out_unlock_pi;

        if (top_waiter != task_top_pi_waiter(task)) {
            if (!detect_deadlock)
                goto out_unlock_pi;

```



```

        else
            requeue = false;
    }
}

// 如果等待者的优先级与任务的优先级相同，则无需进一步调整优先级
if (pi_mutex_waiter_equal(waiter, task_to_waiter(task))) {
    if (!detect_deadlock)
        goto out_unlock_pi;
    else
        requeue = false;
}

// 获取下一个锁
lock = waiter->lock;
// 释放任务的 pi_lock 锁和等待锁
if (!raw_spin_trylock(&lock->wait_lock)) {
    raw_spin_unlock_irq(&task->pi_lock);
    cpu_relax();
    goto retry;
}

if (lock == orig_lock || pi_mutex_owner(lock) == top_task) {
    raw_spin_unlock(&lock->wait_lock);
    ret = -EDEADLK;
    goto out_unlock_pi;
}

if (!requeue) {
    raw_spin_unlock(&task->pi_lock);
    put_task_struct(task);

    if (!pi_mutex_owner(lock)) {
        raw_spin_unlock_irq(&lock->wait_lock);
        return 0;
    }

    task = pi_mutex_owner(lock);
    get_task_struct(task);
    raw_spin_lock(&task->pi_lock);

    next_lock = task_blocked_on_lock(task);
    top_waiter = pi_mutex_top_waiter(lock);

```

```

raw_spin_unlock(&task->pi_lock);
raw_spin_unlock_irq(&lock->wait_lock);

if (!next_lock)
    goto out_put_task;
goto again;
}

prerequeue_top_waiter = pi_mutex_top_waiter(lock);
pi_mutex_dequeue(lock, waiter);

waiter->prio = task->prio;
waiter->deadline = task->dl.deadline;

pi_mutex_enqueue(lock, waiter);

raw_spin_unlock(&task->pi_lock);
put_task_struct(task);

if (!pi_mutex_owner(lock)) {
    if (prerequeue_top_waiter != pi_mutex_top_waiter(lock))
        wake_up_process(pi_mutex_top_waiter(lock)->task);
    raw_spin_unlock_irq(&lock->wait_lock);
    return 0;
}

task = pi_mutex_owner(lock);
get_task_struct(task);
raw_spin_lock(&task->pi_lock);

if (waiter == pi_mutex_top_waiter(lock)) {
    pi_mutex_dequeue_pi(task, prerequeue_top_waiter);
    pi_mutex_enqueue_pi(task, waiter);
    pi_mutex_adjust_prio(task);

} else if (prerequeue_top_waiter == waiter) {
    pi_mutex_dequeue_pi(task, waiter);
    waiter = pi_mutex_top_waiter(lock);
    pi_mutex_enqueue_pi(task, waiter);
    pi_mutex_adjust_prio(task);
} else {
}

```

```

next_lock = task_blocked_on_lock(task);
top_waiter = pi_mutex_top_waiter(lock);

raw_spin_unlock(&task->pi_lock);
raw_spin_unlock_irq(&lock->wait_lock);

if (!next_lock)
    goto out_put_task;

if (!detect_deadlock && waiter != top_waiter)
    goto out_put_task;

goto again;

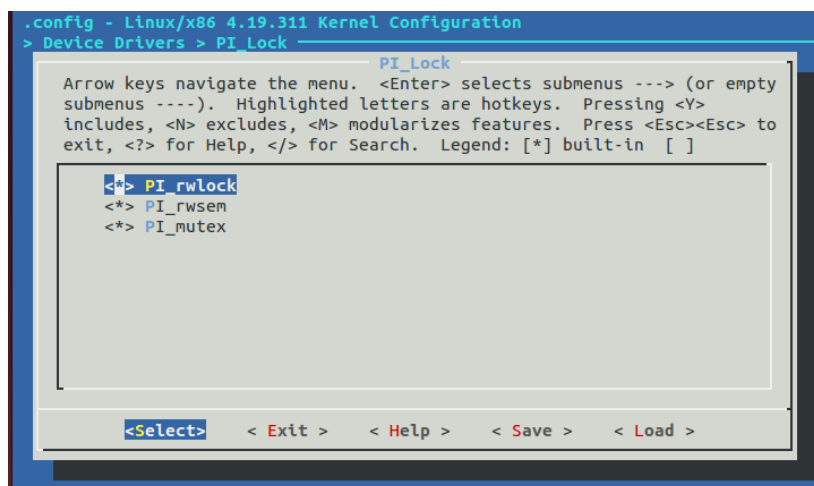
out_unlock_pi:
    raw_spin_unlock_irq(&task->pi_lock);
out_put_task:
    put_task_struct(task);

return ret;
}

```

三、 运行结果演示

本项目基于 Linux Kernel 4.19.311 版本实现，以 Linux 内核模块的形式将添加了优先级继承的 pi_rwlock、pi_rwsem、pi_mutex 添加进内核中，使用 EXPORT_SYMBOL_GPL()方法将其作为符号导出提供给外部模块使用。通过修改 Kconfig 文件将其加入 make menu，在内核编译的时候可以选择是否编译这三个模块，选中的时候如下图所示：



本项目的测试程序使用第一章的例子进行设计，按顺序创建低、高、中三个优先级的线程，低优先级线程先获得锁，然后高优先级线程请求锁被阻塞，中优先级线程尝试抢占低优先级线程的运行。若优先级顺利继承，低优先级线程将继承来自高优先级线程的优先级，不

会被中优先级线程抢占运行，具体代码实现如下：

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>           //kmalloc()
#include <linux/uaccess.h>       //copy_to/from_user()
#include <linux/kthread.h>       //kernel threads
#include <linux/sched.h>         //task_struct
#include <linux/delay.h>
// #include <linux/mutex.h>
#include <linux/pimutex.h>
#include <linux/err.h>

struct pi_mutex etx_pimutex;

unsigned long etx_global_variable = 0;

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);

static struct task_struct *etx_thread_low;
static struct task_struct *etx_thread_high;
static struct task_struct *etx_thread_mid;

/***** Driver functions *****/
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp,
                        char __user *buf, size_t len, loff_t * off);
static ssize_t etx_write(struct file *filp,
                        const char *buf, size_t len, loff_t * off);
/*****/

int thread_function_low(void *pv);
int thread_function_high(void *pv);
int thread_function_mid(void *pv);
```

```

/*
** Thread function 1
*/
int thread_function_low(void *pv)
{
    while(!kthread_should_stop()) {
        pi_mutex_lock(&etx_pmutex);
        pr_info("thread_low start running, get mutex!\n");
        mdelay(10000);
        etx_global_variable++;
        pr_info("In EmbeTronicX Thread Function1 %lu\n", etx_global_variable);
        pi_mutex_unlock(&etx_pmutex);
        pr_info("thread_low stop running, free mutex!\n");
    }
    return 0;
}

/*
** Thread function 2
*/
int thread_function_high(void *pv)
{
    while(!kthread_should_stop()) {
        pr_info("thread_high waiting mutex...\n");
        pi_mutex_lock(&etx_pmutex);
        pr_info("thread_high start running, get mutex!\n");
        etx_global_variable++;
        pr_info("In EmbeTronicX Thread Function_high %lu\n",
etx_global_variable);
        pi_mutex_unlock(&etx_pmutex);
        pr_info("thread_high stop running, free mutex!\n");
        kthread_stop(etx_thread_high);
    }
    return 0;
}

int thread_function_mid(void *pv)
{
    while(!kthread_should_stop()) {
        pr_info("Thread_mid is running!\n");
        mdelay(5000);
    }
}

```

```

        pr_info("Thread_mid running over!\n");
        kthread_stop(etx_thread_mid);
    }
    return 0;
}

//File operation structure
static struct file_operations fops =
{
    .owner      = THIS_MODULE,
    .read       = etx_read,
    .write      = etx_write,
    .open       = etx_open,
    .release    = etx_release,
};

/*
** This function will be called when we open the Device file
*/
static int etx_open(struct inode *inode, struct file *file)
{
    pr_info("Device File Opened...!!!\n");
    return 0;
}

/*
** This function will be called when we close the Device file
*/
static int etx_release(struct inode *inode, struct file *file)
{
    pr_info("Device File Closed...!!!\n");
    return 0;
}

/*
** This function will be called when we read the Device file
*/
static ssize_t etx_read(struct file *filp,
                        char __user *buf, size_t len, loff_t *off)
{
    pr_info("Read function\n");

    return 0;
}

```

```

/*
** This function will be called when we write the Device file
*/
static ssize_t etx_write(struct file *filp,
                        const char __user *buf, size_t len, loff_t *off)
{
    pr_info("Write Function\n");
    return len;
}

/*
** Module Init function
*/
static int __init etx_driver_init(void)
{
    /*Allocating Major number*/
    if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0){
        pr_info("Cannot allocate major number\n");
        return -1;
    }
    pr_info("Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));

    /*Creating cdev structure*/
    cdev_init(&etx_cdev, &fops);

    /*Adding character device to the system*/
    if((cdev_add(&etx_cdev, dev, 1)) < 0){
        pr_info("Cannot add the device to the system\n");
        goto r_class;
    }

    /*Creating struct class*/
    if(IS_ERR(dev_class = class_create(THIS_MODULE, "etx_class"))){
        pr_info("Cannot create the struct class\n");
        goto r_class;
    }

    /*Creating device*/
    if(IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_device"))){
        pr_info("Cannot create the Device \n");
        goto r_device;
    }
}

```

```

pi_mutex_init(&etx_pimutex);

/* Creating Thread 1 */
// etx_thread_low = kthread_run(thread_function_low,NULL,"eTx Thread1");
etx_thread_low = kthread_create(thread_function_low, NULL, "eTx
Thread1");
if(etx_thread_low) {
    kthread_bind(etx_thread_low, 0);
    set_user_nice(etx_thread_low, 15);
    mdelay(2000);
    pr_err("Kthread_low Created Successfully...\n");
    wake_up_process(etx_thread_low);
} else {
    pr_err("Cannot create kthread_low\n");
    goto r_device;
}

/* Creating Thread 2 */
// etx_thread_high = kthread_run(thread_function_high,NULL,"eTx
Thread2");
etx_thread_high = kthread_create(thread_function_high, NULL, "eTx
Thread2");
if(etx_thread_high) {
    kthread_bind(etx_thread_high, 1);
    set_user_nice(etx_thread_high, 5);
    mdelay(2000);
    pr_err("Kthread_high Created Successfully...\n");
    wake_up_process(etx_thread_high);
} else {
    pr_err("Cannot create kthread_high\n");
    goto r_device;
}

etx_thread_mid = kthread_create(thread_function_mid, NULL, "eTx
Thread3");
if(etx_thread_mid) {
    kthread_bind(etx_thread_mid, 0);
    set_user_nice(etx_thread_mid, 10);
    mdelay(2000);
    pr_err("Kthread_mid Created Successfully...\n");
    wake_up_process(etx_thread_mid);
} else {
    pr_err("Cannot create kthread_mid\n");
    goto r_device;
}

```



```

    }

    pr_info("Device Driver Insert...Done!!!\n");
    return 0;

r_device:
    class_destroy(dev_class);
r_class:
    unregister_chrdev_region(dev,1);
    cdev_del(&etx_cdev);
    return -1;
}

/*
** Module exit function
*/
static void __exit etx_driver_exit(void)
{
    kthread_stop(etx_thread_low);
    kthread_stop(etx_thread_high);
    kthread_stop(etx_thread_mid);
    device_destroy(dev_class,dev);
    class_destroy(dev_class);
    cdev_del(&etx_cdev);
    unregister_chrdev_region(dev, 1);
    pr_info("Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_VERSION("1.17");

```

实验结果如下图所示，可见在低优先级线程释放锁后，高优先级线程获得锁继续运行，其中没有被中优先级线程抢占，说明优先级被顺利继承。

```
[ 268.803875] Major = 248 Minor = 0
[ 270.812785] Kthread_low Created Successfully...
[ 270.812910] thread_low start running, get mutex!
[ 272.823757] Kthread_high Created Successfully...
[ 272.823792] thread_high waiting mutex...
[ 274.833683] Kthread_mid Created Successfully...
[ 274.833690] Device Driver Insert...Done!!!
[ 280.854396] In EmbeTronicX Thread Function1 1
[ 280.854408] thread_low stop running, free mutex!
[ 280.854421] thread_high start running, get mutex!
[ 280.854423] In EmbeTronicX Thread Function_high 2
[ 280.854424] thread_high stop running, free mutex!
[ 280.854524] Thread_mid is running!
[ 285.874502] Thread_mid running over!
```

下图是使用原本的没有优先级继承的锁，可见中等优先级线程抢占了低优先级线程，导致高优先级线程在中优先级线程之后运行。

```
[ 767.872613] Major = 248 Minor = 0
[ 767.872793] Kthread_low Created Successfully...
[ 767.872815] thread_low start running, get mutex!
[ 769.880637] Kthread_high Created Successfully...
[ 769.880697] Kthread_mid Created Successfully...
[ 769.880700] Device Driver Insert...Done!!!
[ 772.890959] In EmbeTronicX Thread Function1 1
[ 772.890961] thread_low stop running, free mutex!
[ 787.948038] thread_high waiting mutex...
[ 787.948077] Thread_mid is running!
[ 792.966219] Thread_mid running over!
[ 792.966391] thread_high start running, get mutex!
[ 792.966393] In EmbeTronicX Thread Function_high 2
[ 792.966394] thread_high stop running, free mutex!
```