```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>       //struct file_operations
#include <linux/device.h>   // class_create, device_create
#include <linux/errno.h>    // ERESTART: Interrupted system call should be restarted
#include <asm/uaccess.h>    // copy_to_user()
#include <linux/string.h>   // strncpy()
#include <linux/cdev.h>     // cdev_alloc(), cdev_del(), ...
#include <linux/wait.h>     // wait queues
#include <linux/sched.h>    // TASK_INTERRUPTIBLE used by wake_up_interruptible()

// Metainformation
MODULE_AUTHOR("Stefano Di Martno");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("buffer read write :-P");
MODULE_SUPPORTED_DEVICE("none");

#define MAJORNUM 119
#define NUMDEVICES 1
#define DEVNAME "t12buf"
#define BUFFER_SIZE 10

static struct cdev *cdev = NULL;
static struct class *dev_class;
static char buffer[BUFFER_SIZE];
static int read_position = 0;
static int write_position = 0;
static wait_queue_head_t wq_read;
static wait_queue_head_t wq_write;

// function prototypes
static int __init mod_init(void);
static void __exit mod_exit(void);
static int driver_open(struct inode *inode, struct file *instance);
static ssize_t driver_write(struct file *instanz, const char __user * userbuf,
                size_t count, loff_t * off);
static int driver_close(struct inode *inode, struct file *instance);
static ssize_t driver_read(struct file *file, char *user, size_t count,
                loff_t * offset);

#define free_space() (BUFFER_SIZE - write_position)
#define max_bytes_to_read() (write_position - read_position)

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = driver_read,
    .write = driver_write,
    .open = driver_open,
    .release = driver_close,
};

static int driver_open(struct inode *inode, struct file *instance)
{
    printk("open() called!\n");

    return 0;
}

static int driver_close(struct inode *inode, struct file *instance)
{
    printk("close() called\n");

    return 0;
}

static ssize_t driver_write(struct file *instanz, const char __user * userbuf,
                size_t count, loff_t * off)
{
    ssize_t to_copy;
    char *write_pointer;

    if (free_space() == 0) {
        pr_debug("Producer is going to sleep...\n");
```

```
        if (wait_event_interruptible(wq_write, free_space() > 0))
            return -ERESTART;
    }

    write_pointer = &buffer[write_position];

    if (count < free_space()) {
        to_copy = count;
    } else {
        to_copy = free_space();
    }

    strncpy(write_pointer, userbuf, to_copy);

    write_position += to_copy;

    pr_debug("count: %zu\n", count);
    pr_debug("%zd bytes written\n", to_copy);
    pr_debug("Wake consumer up...\n");

    wake_up_interruptible(&wq_read);

    return to_copy;
}

static ssize_t driver_read(struct file *file, char *user, size_t count,
               loff_t * offset)
{
    long not_copied, to_copy, copied;
    char *read_pointer;

    if (max_bytes_to_read() == 0) {
        pr_debug("Consumer is going to sleep...\n");
        if (wait_event_interruptible(wq_read, max_bytes_to_read() > 0))
            return -ERESTART;
    }

    if (max_bytes_to_read() > count) {
        to_copy = count;
    } else {
        to_copy = max_bytes_to_read();
    }

    read_pointer = &buffer[read_position];

    not_copied = copy_to_user(user, read_pointer, to_copy);
    copied = to_copy - not_copied;

    read_position += copied;

    if (read_position == write_position) {
        read_position = 0;
        write_position = 0;
    }

    pr_debug("read_position %d\n", read_position);
    pr_debug("%ld bytes read\n", copied);

    pr_debug("Wake producer up...\n");
    wake_up_interruptible(&wq_write);

    return copied;
}

static void __exit mod_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
    device_destroy(dev_class, MKDEV(MAJORNUM, 0));
    class_destroy(dev_class);

    if (cdev) {
        cdev_del(cdev);
    }
```

```
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
}

static int __init mod_init(void)
{
    dev_t major_nummer = MKDEV(MAJORNUM, 0);

    printk(KERN_ALERT "Hello, world from buf\n");

    if (register_chrdev_region(major_nummer, NUMDEVICES, DEVNAME)) {
        pr_warn("Device number 0x%x not available ...\n",
            MKDEV(MAJORNUM, 0));
        return -EIO;
    }

    pr_info("Device number 0x%x created\n", MKDEV(MAJORNUM, 0));

    cdev = cdev_alloc();
    if (cdev == NULL) {
        pr_warn("cdev_alloc failed!\n");
        goto free_devnum;
    }

    kobject_set_name(&cdev->kobj, DEVNAME);
    cdev->owner = THIS_MODULE;
    cdev_init(cdev, &fops);

    if (cdev_add(cdev, MKDEV(MAJORNUM, 0), NUMDEVICES)) {
        pr_warn("cdev_add failed!\n");
        goto free_cdev;
    }

    dev_class = class_create(THIS_MODULE, DEVNAME);
    device_create(dev_class, NULL, major_nummer, NULL, DEVNAME);

    init_waitqueue_head(&wq_read);
    init_waitqueue_head(&wq_write);

    return 0;

 free_cdev:
    kobject_put(&cdev->kobj);
    cdev = NULL;
 free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    return -1;
}

module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>       //struct file_operations
#include <linux/device.h>   // class_create, device_create
#include <linux/errno.h>    // ERESTART: Interrupted system call should be restarted
#include <asm/uaccess.h>    // copy_to_user()
#include <linux/string.h>   // strncpy()
#include <linux/cdev.h>     // cdev_alloc(), cdev_del(), ...
#include <linux/wait.h>     // wait queues
#include <linux/sched.h>    // TASK_INTERRUPTIBLE used by wake_up_interruptible()
#include <linux/slab.h>     // kmalloc(), kfree()
#include <linux/kthread.h>
#include <linux/mutex.h>
#include <linux/atomic.h>
#include <linux/workqueue.h>

// Metainformation
MODULE_AUTHOR("Stefano Di Martno");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("buffer read write :-P");
MODULE_SUPPORTED_DEVICE("none");

#define MAJORNUM 100
#define NUMDEVICES 1
#define DEVNAME "t12buf_threaded"

static struct cdev *cdev = NULL;
static struct class *dev_class;

static wait_queue_head_t wq_read;
static wait_queue_head_t wq_write;

struct mutex mutex_buffer;

static struct workqueue_struct *worker_queue;

// function prototypes
static int __init mod_init(void);
static void __exit mod_exit(void);
static int driver_open(struct inode *inode, struct file *instance);
static ssize_t driver_write(struct file *instance, const char __user * userbuf,
                size_t count, loff_t * off);
static int driver_close(struct inode *inode, struct file *instance);
static ssize_t driver_read(struct file *instance, char *user, size_t count,
               loff_t * offset);

#define check_if_thread_is_valid(thread) if(thread == ERR_PTR(-ENOMEM)) \
        { \
                pr_crit("thread could not be created!\n"); \
                return -EIO; \
        }

#define check_memory(pointer) if (pointer == NULL) {\
         pr_alert("Could not allocate memory!\n");\
         return -1;\
         }

typedef struct {
    size_t count;
    struct task_struct *thread_write;
    char *user;
} write_data;

typedef struct {
    atomic_t wake_up;
    wait_queue_head_t wait_queue;
    char *user;
} read_data;

typedef struct {
    struct work_struct work;
    write_data *write_data;
    read_data *read_data;
```

```
    struct completion on_exit;
    int ret;
} private_data;

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = driver_read,
    .write = driver_write,
    .open = driver_open,
    .release = driver_close,
};

#define stackTotalSize 10

typedef struct {
    void *stack;        /* Points to the objects on the stack */
    size_t stack_size;  /* Element size of the element type */
    int current_size;   /* The real amount of objects on the stack */
    struct mutex mutex;
    void (*freefn) (const void *);  /* free function for more complex data types */
} genstack;

static genstack stack;

void init_genstack(genstack * s, size_t stack_size,
           void (*freefn) (const void *))
{
    /* Default initialization */
    s->stack = kmalloc(stackTotalSize * stack_size, GFP_KERNEL);

    if (s->stack == NULL) {
        pr_alert("Could not init stack!\n");
        return;
    }

    mutex_init(&s->mutex);
    s->stack_size = stack_size;
    s->current_size = 0;
    s->freefn = freefn;
}

int genstack_push(genstack * s, const void *elem_addr)
{
    char *ptarget_addr;

    mutex_lock(&s->mutex);

    if (s->current_size == stackTotalSize) {
        mutex_unlock(&s->mutex);
        return -1;
    }

    /* Equivalent to &s->stack[s->current_size] */
    ptarget_addr = (char *)s->stack + s->current_size * s->stack_size;

    memcpy(ptarget_addr, elem_addr, s->stack_size);
    s->current_size++;

    mutex_unlock(&s->mutex);

    return 0;
}

void genstack_pop(genstack * s, void *elem_addr)
{
    char *pSourceAddr;

    mutex_lock(&s->mutex);

    /* Equivalent to &s->stack[s->current_size - 1] */
    pSourceAddr = (char *)s->stack + (s->current_size - 1) * s->stack_size;

    memcpy(elem_addr, pSourceAddr, s->stack_size);
```

```
    s->current_size--;

    mutex_unlock(&s->mutex);
}

int genstack_empty(const genstack * s)
{
    return s->current_size == 0;
}

int genstack_full(const genstack * s)
{
    return s->current_size == stackTotalSize;
}

void genstack_dispose(genstack * s)
{
    if (s->freefn != NULL && s->current_size > 0) {
        char *pSourceAddr;

        for (; s->current_size > 0; s->current_size--) {
            /* Equivalent to &s->stack[s->current_size - 1] */
            pSourceAddr =
                (char *)s->stack + (s->current_size -
                        1) * s->stack_size;

            /* call free function of the client */
            s->freefn(pSourceAddr);
        }
    }

    mutex_destroy(&s->mutex);
    kfree(s->stack);
    s->stack = NULL;
}

static int thread_write(void *write_data)
{
    private_data *data = (private_data *) write_data;

    if (genstack_full(&stack))  // For debug added
    {
        pr_debug("Producer is going to sleep...\n");
        if (wait_event_interruptible(wq_write, !genstack_full(&stack)))
            return -ERESTART;
    }

    genstack_push(&stack, &data->write_data->user);

    complete_and_exit(&data->on_exit, 0);
}

static void thread_read(struct work_struct *work)
{
    private_data *data = container_of(work, private_data, work);

    if (genstack_empty(&stack)) // For debug added
    {
        pr_debug("Consumer is going to sleep...\n");
        if (wait_event_interruptible(wq_read, !genstack_empty(&stack))) {
            data->ret = -ERESTART;
            return;
        }
    }

    genstack_pop(&stack, &data->read_data->user);

    data->ret = strlen(data->read_data->user);

    pr_debug("Wake producer and read() up...\n");

    atomic_set(&data->read_data->wake_up, 1);
```

```
    wake_up_interruptible(&data->read_data->wait_queue);
    wake_up_interruptible(&wq_write);
}

static int driver_open(struct inode *inode, struct file *instance)
{
    private_data *data;

    printk("open() called!\n");

    data = (private_data *) kmalloc(sizeof(private_data), GFP_KERNEL);
    check_memory(data);

    init_completion(&(data->on_exit));

    // Do only kmalloc() on read() and write, if write_data or read_data are NULL!
    data->write_data = NULL;
    data->read_data = NULL;

    instance->private_data = data;

    return 0;
}

static int driver_close(struct inode *inode, struct file *instance)
{
    private_data *data = (private_data *) instance->private_data;

    printk("close() called\n");

    if (data->write_data != NULL) {
        kfree(data->write_data);
    }

    if (data->read_data != NULL) {
        kfree(data->read_data);
    }

    kfree(data);

    return 0;
}

static ssize_t driver_write(struct file *instance, const char __user * userbuf,
                size_t count, loff_t * off)
{
    private_data *data = (private_data *) instance->private_data;

    if (data->write_data == NULL) {
        data->write_data =
            (write_data *) kmalloc(sizeof(write_data), GFP_KERNEL);
        check_memory(data->write_data);

        pr_debug("Create producer thread for the first time...\n");
    }

    data->write_data->count = count;
    pr_debug("Write: Call wake_up_process()\n");

    data->write_data->thread_write =
        kthread_create(thread_write, data, "thread_write");
    check_if_thread_is_valid(data->write_data->thread_write);

    pr_debug("Write: bytes to copy: %d\n", count);

    data->write_data->user = kmalloc(count + 1, GFP_KERNEL);
    check_memory(data->write_data->user);

    if (copy_from_user(data->write_data->user, userbuf, count) != 0) {
        pr_crit("Could not copy from user space!!!\n");
        return -1;
    }
```

```
    data->write_data->user[count] = '\0';

    wake_up_process(data->write_data->thread_write);
    wait_for_completion(&data->on_exit);

    pr_debug("Wake consumer up...\n");

    wake_up_interruptible(&wq_read);

    return count;
}

static ssize_t driver_read(struct file *instance, char *user, size_t count,
               loff_t * offset)
{
    unsigned long not_copied, to_copy, copied;
    private_data *data = (private_data *) instance->private_data;
    wait_queue_head_t *wait_queue;
    atomic_t *wake_up;

    if (data->read_data == NULL) {
        data->read_data =
            (read_data *) kmalloc(sizeof(read_data), GFP_KERNEL);
        check_memory(data->read_data);

        init_waitqueue_head(&data->read_data->wait_queue);
        INIT_WORK(&data->work, thread_read);
        pr_debug("Create consumer thread for the first time...\n");
    }
    // Init read_data
    wait_queue = &data->read_data->wait_queue;
    wake_up = &data->read_data->wake_up;

    atomic_set(wake_up, 0);

    if (!queue_work(worker_queue, &data->work)) {
        pr_crit("queue_work not successful ...\n");
    }

    if (wait_event_interruptible(*wait_queue, atomic_read(wake_up)))
        return -ERESTART;

    if (data->ret < 0) {
        return data->ret;
    }

    to_copy = data->ret;
    not_copied = copy_to_user(user, data->read_data->user, to_copy);
    copied = to_copy - not_copied;

    pr_debug("not_copied: %lu to_copy: %lu. count %d. %lu bytes read\n",
         not_copied, to_copy, count, copied);

    kfree(data->read_data->user);

    return copied;
}

static void __exit mod_exit(void)
{
    printk(KERN_ALERT "buf_threaded: Goodbye, cruel world\n");
    device_destroy(dev_class, MKDEV(MAJORNUM, 0));
    class_destroy(dev_class);

    if (cdev) {
        cdev_del(cdev);
    }

    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);

    if (worker_queue) {
        destroy_workqueue(worker_queue);
        pr_debug("workqueue destroyed\n");
```

```
    }

    genstack_dispose(&stack);
}

static int __init mod_init(void)
{
    dev_t major_nummer = MKDEV(MAJORNUM, 0);

    printk(KERN_ALERT "buf_threaded: Hello, world!\n");

    if (register_chrdev_region(major_nummer, NUMDEVICES, DEVNAME)) {
        pr_warn("Device number 0x%x not available ...\n",
            MKDEV(MAJORNUM, 0));
        return -EIO;
    }

    pr_info("Device number 0x%x created\n", MKDEV(MAJORNUM, 0));

    cdev = cdev_alloc();

    if (cdev == NULL) {
        pr_warn("cdev_alloc failed!\n");
        goto free_devnum;
    }

    kobject_set_name(&cdev->kobj, DEVNAME);
    cdev->owner = THIS_MODULE;
    cdev_init(cdev, &fops);

    if (cdev_add(cdev, MKDEV(MAJORNUM, 0), NUMDEVICES)) {
        pr_warn("cdev_add failed!\n");
        goto free_cdev;
    }

    dev_class = class_create(THIS_MODULE, DEVNAME);
    device_create(dev_class, NULL, major_nummer, NULL, DEVNAME);

    init_genstack(&stack, sizeof(char **), kfree);

    init_waitqueue_head(&wq_read);
    init_waitqueue_head(&wq_write);

    worker_queue = create_singlethread_workqueue("bufThread");

    return 0;

 free_cdev:
    kobject_put(&cdev->kobj);
    cdev = NULL;
 free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    return -1;
}

module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>     // kmalloc(), kfree()
#include <asm/uaccess.h>    // copy_to_user()

MODULE_AUTHOR("Jakub Werner");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A dummy driver");
MODULE_SUPPORTED_DEVICE("none");

#define MAJORNUM 120
#define NUMDEVICES 2
#define DEVNAME "t12hello2"

#define HELLO  "Hello World!!!\n"

static struct cdev *cdev = NULL;

static int is_open = 0;

static struct class *dev_class;
static struct device *device;

static ssize_t driver_read(struct file *instanz, char *buf, size_t num,
                loff_t * off);
static ssize_t driver_open(struct inode *inode, struct file *file);
static ssize_t driver_close(struct inode *inode, struct file *file);

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = driver_read,
    .open = driver_open,
    .release = driver_close
};

typedef struct read_cout_t {
    ssize_t count;
} read_count;

static read_count *rc;

static int __init mod_init(void)
{
    dev_t major_nummer = MKDEV(MAJORNUM, 0);

    if (register_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES, DEVNAME)) {
        pr_warn("Device number 0x%x not available ...\n",
            MKDEV(MAJORNUM, 0));
        return -EIO;
    }

    pr_info("Device number 0x%x created\n", MKDEV(MAJORNUM, 0));

    cdev = cdev_alloc();
    if (cdev == NULL) {
        pr_warn("cdev_alloc failed!\n");
        goto free_devnum;
    }

    kobject_set_name(&cdev->kobj, DEVNAME);
    cdev->owner = THIS_MODULE;
    cdev_init(cdev, &fops);

    if (cdev_add(cdev, MKDEV(MAJORNUM, 0), NUMDEVICES)) {
        pr_warn("cdev_add failed!\n");
        goto free_cdev;
    }

    device = device;
```

```
    dev_class = class_create(THIS_MODULE, DEVNAME);
    device = device_create(dev_class, NULL, major_nummer, NULL, DEVNAME);

    // Device Specific operations.

    return 0;

 free_cdev:
    kobject_put(&cdev->kobj);
    cdev = NULL;
 free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    return -1;
}

static ssize_t driver_read(struct file *instanz, char *userbuf, size_t count,
               loff_t * off)
{

    ssize_t len = strlen(HELLO);
    if (rc->count < len) {
        rc->count += len - copy_to_user(userbuf, HELLO, len);
        pr_debug
            ("Module fops : sent %d bytes to user space and String-len was: %d \n",
             rc->count, len);
        return rc->count;
    }
    return 0;

}

static ssize_t driver_open(struct inode *inode, struct file *file)
{

    if (is_open)
        return -EBUSY;

    rc = kmalloc(sizeof(read_count), GFP_USER);
    rc->count = 0;

    is_open++;

    try_module_get(THIS_MODULE);
    pr_debug
        ("Module fops:device %s was opened from device with minor no %d \n",
         DEVNAME, iminor(inode));
    return 0;

}

static ssize_t driver_close(struct inode *inode, struct file *file)
{
    is_open--;

    module_put(THIS_MODULE);
    pr_debug("Module fops:device %s was closed \n", DEVNAME);
    kfree(rc);
    return 0;

}

static void __exit mod_exit(void)
{

    if (cdev) {
        cdev_del(cdev);
    }

    device_destroy(dev_class, MKDEV(MAJORNUM, 0));
    class_destroy(dev_class);

    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    printk(KERN_ALERT "Goodbye, cruel world\n");
```

```
}

module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/completion.h>
#include <linux/sched.h> // TASK_INTERRUPTIBLE used by wake_up_interruptible()
#include <linux/kthread.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stefano Di Martno");
MODULE_DESCRIPTION("kthread sample");
MODULE_SUPPORTED_DEVICE("none");

static struct task_struct *thread_id;
static wait_queue_head_t wq;
static DECLARE_COMPLETION(on_exit);

static int thread_code(void *data)
{
    unsigned long timeout;

    //daemonize("MySySoKThread");
    allow_signal(SIGTERM);

    while (kthread_should_stop() == 0)
    {
        timeout = 2 * HZ; // wait 2 second
        timeout = wait_event_interruptible_timeout(wq, (timeout == 0), timeout);

        pr_debug("thread_code: woke up ...\n");

        if(timeout == -ERESTARTSYS)
        {
            pr_info("got signal, break\n");
            break;
        }
    }

    complete_and_exit(&on_exit, 0);
}

static int __init kthread_init(void)
{
    printk(KERN_ALERT "kthread: Hello, world\n");

    init_waitqueue_head(&wq);
    thread_id = kthread_create(thread_code, NULL, "MySySoKThread");

    if(thread_id == ERR_PTR(-ENOMEM))
    {
        pr_crit("kthread could not be created!\n");
        return -EIO;
    }

    wake_up_process(thread_id);

    return 0;
}

static void __exit kthread_exit(void)
{
    kill_pid(task_pid(thread_id), SIGTERM, 1);
    wait_for_completion(&on_exit);

    printk(KERN_ALERT "kthread: Goodbye, cruel world\n");
}

module_init(kthread_init);
module_exit(kthread_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/delay.h>
#include <linux/semaphore.h>
#include <linux/timer.h>
#include <linux/slab.h>     // kmalloc(), kfree()
#include <asm/uaccess.h>    // copy_to_user()

MODULE_AUTHOR("Stefano Di Martino");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("lock");

#define MAJORNUM 101
#define NUMDEVICES 1
#define DEVNAME "t12lock"
#define DEFAULT_SLEEP_TIME_SECONDS 10
#define DEFAULT_SLEEP_TIME_MSECONDS 200

static struct cdev *cdev = NULL;

static struct class *dev_class;
static struct device *device;

static ssize_t driver_open(struct inode *inode, struct file *file);
static ssize_t driver_close(struct inode *inode, struct file *file);
static ssize_t driver_read(struct file *instance, char *user, size_t count,
               loff_t * offset);

static int MODULE_EXIT = 0;

DEFINE_SEMAPHORE(open_once);

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = driver_open,
    .read = driver_read,
    .release = driver_close
};

static ssize_t driver_read(struct file *instance, char *user, size_t count,
               loff_t * offset)
{
    return 0;
}

static int __init mod_init(void)
{
    dev_t major_nummer = MKDEV(MAJORNUM, 0);

    printk(KERN_ALERT "lock: Hello, world!\n");

    if (register_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES, DEVNAME)) {
        pr_warn("Device number 0x%x not available ...\n",
            MKDEV(MAJORNUM, 0));
        return -EIO;
    }

    pr_info("Device number 0x%x created\n", MKDEV(MAJORNUM, 0));

    cdev = cdev_alloc();
    if (cdev == NULL) {
        pr_warn("cdev_alloc failed!\n");
        goto free_devnum;
    }

    kobject_set_name(&cdev->kobj, DEVNAME);
    cdev->owner = THIS_MODULE;
    cdev_init(cdev, &fops);

    if (cdev_add(cdev, MKDEV(MAJORNUM, 0), NUMDEVICES)) {
```

```
        pr_warn("cdev_add failed!\n");
        goto free_cdev;
    }

    device = device;

    dev_class = class_create(THIS_MODULE, DEVNAME);
    device = device_create(dev_class, NULL, major_nummer, NULL, DEVNAME);

    // Device Specific operations.

    return 0;

 free_cdev:
    kobject_put(&cdev->kobj);
    cdev = NULL;
 free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    return -1;
}

static ssize_t driver_open(struct inode *inode, struct file *file)
{
    pr_debug
        ("Module fops: device %s was opened from device with minor no %d\n",
         DEVNAME, iminor(inode));

    if (!down_trylock(&open_once)) {
        pr_debug("Module fops: device %s got lock falling to sleep\n",
            DEVNAME);
    } else {
        unsigned long start = jiffies;

        while (down_trylock(&open_once) && !MODULE_EXIT) {
            pr_debug
                ("Module fops: device %s waited %d msecs till semaphore is unlocked
\n",
                 DEVNAME, jiffies_to_msecs(jiffies - start));
            msleep(DEFAULT_SLEEP_TIME_MSECONDS);
        }

        pr_debug
            ("Module fops:device %s got lock after waiting %d semaphore falling to s
leep\n",
             DEVNAME, jiffies_to_msecs(jiffies - start));
    }

    ssleep(DEFAULT_SLEEP_TIME_SECONDS);

    return 0;
}

static ssize_t driver_close(struct inode *inode, struct file *file)
{
    up(&open_once);

    pr_debug("Module fops: device %s was closed \n", DEVNAME);
    return 0;
}

static void __exit mod_exit(void)
{
    if (cdev) {
        cdev_del(cdev);
    }

    MODULE_EXIT = 1;

    device_destroy(dev_class, MKDEV(MAJORNUM, 0));
    class_destroy(dev_class);
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    printk(KERN_ALERT "lock: Goodbye, cruel world\n");
}
```

```
module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/delay.h>
#include <linux/mutex.h>
#include <linux/timer.h>
#include <linux/slab.h>     // kmalloc(), kfree()
#include <asm/uaccess.h>    // copy_to_user()

MODULE_AUTHOR("Jakub Werner");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("mutex");

#define MAJORNUM 125
#define NUMDEVICES 2
#define DEVNAME "t12mutex"
#define DEFAULT_SLEEP_TIME_SECONDS 10
#define DEFAULT_SLEEP_TIME_MSECONDS 200

static struct cdev *cdev = NULL;

static struct class *dev_class;
static struct device *device;

static ssize_t driver_open(struct inode *inode, struct file *file);
static ssize_t driver_close(struct inode *inode, struct file *file);
static ssize_t driver_read(struct file *instance, char *user, size_t count,
               loff_t * offset);

static int MODULE_EXIT = 0;

DEFINE_MUTEX(open_once);

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = driver_open,
    .read = driver_read,
    .release = driver_close
};

static ssize_t driver_read(struct file *instance, char *user, size_t count,
               loff_t * offset)
{
    return 0;
}

static int __init mod_init(void)
{
    dev_t major_nummer = MKDEV(MAJORNUM, 0);

    if (register_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES, DEVNAME)) {
        pr_warn("Device number 0x%x not available ...\n",
            MKDEV(MAJORNUM, 0));
        return -EIO;
    }

    pr_info("Device number 0x%x created\n", MKDEV(MAJORNUM, 0));

    cdev = cdev_alloc();
    if (cdev == NULL) {
        pr_warn("cdev_alloc failed!\n");
        goto free_devnum;
    }

    kobject_set_name(&cdev->kobj, DEVNAME);
    cdev->owner = THIS_MODULE;
    cdev_init(cdev, &fops);

    if (cdev_add(cdev, MKDEV(MAJORNUM, 0), NUMDEVICES)) {
        pr_warn("cdev_add failed!\n");
        goto free_cdev;
```

```
    }

    device = device;

    dev_class = class_create(THIS_MODULE, DEVNAME);
    device = device_create(dev_class, NULL, major_nummer, NULL, DEVNAME);

    // Device Specific operations.

    return 0;

 free_cdev:
    kobject_put(&cdev->kobj);
    cdev = NULL;
 free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    return -1;
}

static ssize_t driver_open(struct inode *inode, struct file *file)
{
    pr_debug
        ("Module fops:device %s was opened from device with minor no %d\n",
         DEVNAME, iminor(inode));

    if (mutex_trylock(&open_once)) {
        pr_debug("Module fops:device %s got lock falling to sleep\n",
             DEVNAME);
        ssleep(DEFAULT_SLEEP_TIME_SECONDS);
    } else {
        unsigned long start = jiffies;
        while (!mutex_trylock(&open_once) && !MODULE_EXIT) {
            pr_debug
                ("Module fops:device %s waited %d msecs till mutex is unlocked \n",
                 DEVNAME, jiffies_to_msecs(jiffies - start));
            msleep(DEFAULT_SLEEP_TIME_MSECONDS);
        }
        pr_debug
            ("Module fops:device %s got lock after waiting %d msecs falling to sleep
\n",
             DEVNAME, jiffies_to_msecs(jiffies - start));
        ssleep(DEFAULT_SLEEP_TIME_SECONDS);
    }
    return 0;
}

static ssize_t driver_close(struct inode *inode, struct file *file)
{

    mutex_unlock(&open_once);

    pr_debug("Module fops:device %s was closed \n", DEVNAME);
    return 0;

}

static void __exit mod_exit(void)
{

    if (cdev) {
        cdev_del(cdev);
    }

    MODULE_EXIT = 1;

    if (mutex_is_locked(&open_once)) {
        printk(KERN_ALERT "Mutex is still locked\n");
        mutex_unlock(&open_once);

    }
    mutex_destroy(&open_once);

    device_destroy(dev_class, MKDEV(MAJORNUM, 0));
```

```
    class_destroy(dev_class);
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>     // kmalloc(), kfree()
#include <asm/uaccess.h>    // copy_to_user()

MODULE_AUTHOR("Jakub Werner");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A dummy driver");
MODULE_SUPPORTED_DEVICE("none");

#define MAJORNUM 121
#define NUMDEVICES 2
#define DEVNAME "t12tasklet"

#define TASKLET_CALLED "My Tasklet Was Called"

static struct cdev *cdev = NULL;

static struct class *dev_class;
static struct device *device;

static int is_open = 0;

static atomic_t v;

static ssize_t driver_open(struct inode *inode, struct file *file);
static ssize_t driver_close(struct inode *inode, struct file *file);

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = driver_open,
    .release = driver_close,
};

static char tasklet_called[] = "my tasklet was called";

static void tasklet_function(unsigned long data);

DECLARE_TASKLET(tasklet, tasklet_function, (unsigned long)&tasklet_called);

static void tasklet_function(unsigned long data)
{
    printk("%s\n", (char *)data);
}

static int __init mod_init(void)
{
    dev_t major_nummer = MKDEV(MAJORNUM, 0);

    atomic_set(&v, -1);

    if (register_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES, DEVNAME)) {
        pr_warn("Device number 0x%x not available ...\n",
            MKDEV(MAJORNUM, 0));
        return -EIO;
    }

    pr_info("Device number 0x%x created\n", MKDEV(MAJORNUM, 0));

    cdev = cdev_alloc();
    if (cdev == NULL) {
        pr_warn("cdev_alloc failed!\n");
        goto free_devnum;
    }

    kobject_set_name(&cdev->kobj, DEVNAME);
    cdev->owner = THIS_MODULE;
    cdev_init(cdev, &fops);
```

```
    if (cdev_add(cdev, MKDEV(MAJORNUM, 0), NUMDEVICES)) {
        pr_warn("cdev_add failed!\n");
        goto free_cdev;
    }

    device = device;

    dev_class = class_create(THIS_MODULE, DEVNAME);
    device = device_create(dev_class, NULL, major_nummer, NULL, DEVNAME);

    return 0;

 free_cdev:
    kobject_put(&cdev->kobj);
    cdev = NULL;
 free_devnum:
    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    return -1;
}

static ssize_t driver_open(struct inode *inode, struct file *file)
{
    tasklet_schedule(&tasklet);
    is_open++;
    try_module_get(THIS_MODULE);
    pr_debug
        ("Module fops:device %s was opened from device with minor no %d \n",
         DEVNAME, iminor(inode));
    return 0;
}

static ssize_t driver_close(struct inode *inode, struct file *file)
{
    is_open--;
    module_put(THIS_MODULE);
    pr_debug("Module fops:device %s was closed \n", DEVNAME);
    return 0;
}

static void __exit mod_exit(void)
{

    if (cdev) {
        cdev_del(cdev);
    }

    device_destroy(dev_class, MKDEV(MAJORNUM, 0));
    class_destroy(dev_class);

    tasklet_kill(&tasklet);
    tasklet_disable(&tasklet);

    unregister_chrdev_region(MKDEV(MAJORNUM, 0), NUMDEVICES);
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/timer.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>     // kmalloc(), kfree()
#include <asm/uaccess.h>    // copy_to_user()

MODULE_AUTHOR("Jakub Werner");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Timer");
MODULE_SUPPORTED_DEVICE("none");

#define DEVNAME "t12timer"

static struct timer_list timer;

static void timer_callback(unsigned long data);

typedef struct timer_data_t {
    unsigned long jiffies_stamp;
} timer_data;

static int __init mod_init(void)
{
    timer_data *data;

    data = kmalloc(sizeof(timer_data), GFP_KERNEL);
    data->jiffies_stamp = 0;
    setup_timer(&timer, timer_callback, (unsigned long)data);
    mod_timer(&timer, jiffies + msecs_to_jiffies(2000));

    return 0;

}

static void timer_callback(unsigned long data)
{
    timer_data *d = (timer_data *) data;
    unsigned long time_diff = jiffies - d->jiffies_stamp;
    unsigned long min = 0, max = 0;

    min = min(min, time_diff);
    max = max(max, time_diff);

    if (d->jiffies_stamp)
        printk(DEVNAME
               " called at (%ld) time since the last call = %ld, clockcyles min: %ld
, max %ld.\n",
               jiffies / HZ, time_diff / HZ, min / HZ, max / HZ);
    else
        printk(DEVNAME " called first time (%ld).\n", jiffies);

    mod_timer(&timer, jiffies + msecs_to_jiffies(2000));
    d->jiffies_stamp = jiffies;
}

static void __exit mod_exit(void)
{

    int ret;

    ret = del_timer(&timer);
    if (ret)
        printk("The timer is still in use...\n");
}

module_init(mod_init);
module_exit(mod_exit);
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/timer.h>
#include <linux/delay.h>
#include <linux/sched.h>
#include <linux/slab.h>     // kmalloc(), kfree()

MODULE_AUTHOR("Jakub Werner");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Workqueue");

#define DEVNAME "t12workqueue"
#define WORKQUEUE_SIZE 10

static void workqueue_callback(struct work_struct *data);

typedef struct timer_data_t {
    unsigned long jiffies_stamp;
} timer_data;

typedef struct {
    struct work_struct work;
    timer_data *timer;
    int work_number;
} work_data;

static struct workqueue_struct *wq;

static work_data *work[WORKQUEUE_SIZE];

static int __init mod_init(void)
{

    int i = 0;

    wq = create_workqueue("my_workqueue");
    if (wq) {
        for (i = 0; i < WORKQUEUE_SIZE; i++) {
            work[i] =
                (work_data *) kmalloc(sizeof(work_data),
                          GFP_KERNEL);
            if (work[i]) {
                INIT_WORK((struct work_struct *)work[i],
                      workqueue_callback);
                work[i]->timer =
                    (timer_data *) kmalloc(sizeof(timer_data),
                                GFP_KERNEL);
                if (work[i]->timer) {
                    work[i]->timer->jiffies_stamp = 0;
                    work[i]->work_number = i;
                    queue_work(wq,
                           (struct work_struct *)
                           work[i]);
                }
            }
        }

    }

    return 0;
}

static void workqueue_callback(struct work_struct *data)
{
    work_data *my_data = (work_data *) data;
    timer_data *d = my_data->timer;
    unsigned long time_diff = jiffies - d->jiffies_stamp;
    unsigned long min = 0, max = 0;

    min = min(min, time_diff);
    max = max(max, time_diff);
```

```
    if (d->jiffies_stamp)
        printk(DEVNAME
               "-%d called at (%ld) time since the last call = %ld, clockcyles min:
%ld, max %ld.\n",
               my_data->work_number, jiffies / HZ, time_diff / HZ,
               min / HZ, max / HZ);
    else
        printk(DEVNAME "-%d called first time (%ld).\n",
               my_data->work_number, jiffies);

    d->jiffies_stamp = jiffies;

    ssleep(2);
    queue_work(wq, (struct work_struct *)data);

    return;
}

static void __exit mod_exit(void)
{

    int i = 0;
    pr_debug("mod_exit called\n");
    for (i = 0; i < WORKQUEUE_SIZE; i++) {
        cancel_work_sync(&(work[i]->work));
    }
    flush_scheduled_work();
    flush_workqueue(wq);
    destroy_workqueue(wq);

    return;

}

module_init(mod_init);
module_exit(mod_exit);
```