University of Southern Denmark

DM510 Assignment 3

Kernel Module

Author:
Sebastian Eklund Larsen
<selar16>

April 23, 2018

Contents

1 Introduction

In this assignment a module for kernel 4.15 has to be made. The Module Should allow for the transfer of data detween the devices /dev/dm510-0 and /dev/dm510-1, through two buffers. Each device can only read from one buffer and write to the other buffer, dm510-0 reads from buffer 0 and writes to buffer 1, whereas dm510-1 does the opposite. The module should either use a major number of 254 or dynamically allocate the number. The module also has to support blocking and non-blocking I/O, and have simple device control in form of ioctl system calls for adjusting buffer sizes and setting the maximum number of readers who are allowed to read the device at a time. A final requirement is that many readers may open the file at the same time, but only one may be opened for writing at a time.

2 Design

A device has a structure, which holds pointers to the structures of the buffers it can read or write to/from. The reason why the devices do not directly point to the bufferspace, is because each buffer needs multiple variables, which would be better defined in a new structure. The buffers have been given their own structure, which contain the actual pointer to the bufferspace. Inside this structure is also a mutex to make sure that any changes to the buffer structure, does not get invalidated through race conditions.

The bufferspace is not allocated in init, and is instead allocated in open if the bufferpointer does not exist. The bufferspace is also freed when no readers or writers are present. This is done to allow the buffer size to be able to be changed, and applied when a buffer is able to be allocated again, since it can be assumed that the data is important

3 Implementation

3.1 Init

The Init function starts by making a dev of devt type, it does this by checking if a major number is already defined for the module, in which case it creates the device from the major number and registers the device(s). If a major

number was not set or is set to 0, then init dynamically allocates the device(s), and dev gets assigned major number which was free.

After having registered the devices, the space for the devices structures is allocated, returning an error if unsuccessful. Now the spaces for the two buffers are allocates, and their mutex's and wait queues are initialized. at the end of the init function the pointers in the devices structures are set and their cdev variable is setup. if init was succesful it will return 0, otherwise it will return an error code equal to that of whichever function did not complete as intended.

3.2 Cleanup

in the cleanup function the allocated space from the devices has to be freed. for the buffers inside the devices only one device has to be looked through, since they both point to each of the buffers. Once the buffers have been freed the cdevs are deleted, and the pointer to the devices freed, Whereafter the devices device is unregistered.

3.3 Open

Open starts by finding the device associated with the inode pointer, and the stores it in the private data of the file pointer for easy access if needed later. if the file was opened in read mode then open tries to acquire the lock on the devices read buffer. Whereafter it checks if the bufferspace exists, and if not it allocates some, according to the size defined. It then increments the number of readers on the buffer, and forfeits the lock. In read mode it also checks if a max number of readers have been set, blocking if the number equal to or less than nreaders. If the limit is set to 0 it is treated as infinity.

If the file was opened in write mode then the same thing happens for the write buffer instead, but a block is created to to block the open if there are more than 0 writers on the file. In order to support non-blocking I/O a return of an error code -EAGAIN is returned if the non blocking flag is set. if the flag is not set then the process is added to the wait queue for writers in its writer buffer, and waits for the event where nwriters is 0, adding the process to the read buffers wait queue.

3.4 Release

The release function finds the device which will be used by using the private data of the file pointer, which is where we saved it in open. If the file was in read mode, the read buffers lock will be aquired, whereafter the nreaders of the devices read buffer is decremented, and if there are no readers or writers in the buffer, then the bufferspace is freed. Once it is done with the reader buffer it unlocks the mutex. Afterwards it also wakes up the readers on its read buffer, as it is passible that the processes blocked by reader limit can do something now.

If the file was in write mode then it does the same, but for the writer buffer, one added thing which happens when a writer is released, is that the queue for writers in the writer buffer is awoken. This is done because the number of writers should be 0 now, which would allow for a new writer to open the file.

3.5 Read

The read function tries to obtain the lock for the read buffer, of the device associated with the file pointer. The code prepares to block if the read pointer is equal to the write pointer, by adding it to the read wait queue in the read buffer. If the non blocking flag is set it instead returns an error. Once read no longer waits, It checks whether or not the write pointer has wrapped back behind the read pointer, if it has then it will either read as much as requested or until the end of the buffer, whichever is shortest. If the pointer has not wrapped then it will at most write the difference between the reader and writer pointer. copytouser is used to copy from the buffer in the module to the users, it can copy the amount which we found, and it returns an error if the copy failed. once read the read pointer is incremented by the amount read, set to the beginning of the buffer if it reaches the end. Once the read is done, it wakes the writers associated with the devices reader buffer, as the buffer now is in a state where it is not full.

3.6 Write

Write starts by obtaining the lock for the devices writer buffer. It then checks if the buffer is full, if it is the function blocks, returning an error if non block is set. In order to help with finding the free writespace a helper function

spacefree is create, this function returns amount of consective space can be written. If the the writer pointer is equal to the read pointer then the buffer buffer is empty and buffersize - 1 can be written, because all of bufferspace would make it loop back to the writer pointer and make it seem like the buffer was empty. If the pointers are not equal the free space is defined as the amount of spaces for the writer pointer to reach the read pointer minus 1.

Once the write no longer blocks, it checks whether the requested amount is more the free space returned from spacefree, if spacefree returned a lower number, it will be used as the amount to be written going forward. If the writer has not wrapped the amount to be written is either set to stay the same or change to the difference between the writer pointer and the pointer to the end, whichever is lowest. If the writer has wrapped then the amount to write changes to the difference between the writer pointer and the reader pointer - 1, or stays as it is, depending on which is lowest.

Once the amount to write has been found, copyfromuser is used to copy the data to be written into the writer buffer. Afterwards the writer pointer is incremented with the same amount as was written, and if the pointer reached the end of the buffer it is set to the start of the buffer. Once write lets go of its lock it wakes the in queue of the writer buffer, as its readers may have been sleeping when the buffer was empty, which it may no longer be now that data has been written to it.

3.7 ioctl

In ioctl there are commands defined for changing the maximum readers the module will allow on a device, and changing the buffer size according to the argument.

4 Sleep

Sleeping has been used in the places where processes needed to block. Readers sleep when the buffer is empty and when nreaders limit has been reached, and are awoken when a write has finished and when readers release the file. Writers sleep when the file is full and when a writer releases the file. These waits are implemented by using the wait queues in the buffer structures.

Since readers block without anything in the buffer, it is possiple that they

will block forever unless a writer appears and writes to the buffer. This is still in line with the specifications of the module, but it could create problems if used incorrectly.

5 Tests

The test consists of loading the module into the kernel 4.15, and then running test, which was a test provided on the course page. This test creates two processes where one first tries to write to the file and the other first tries to read from the file. afterwards the same is done but in the where they do the opposite. The module was able to reurn the same value as was written to it, which proves that the module can read and write properly. It also shows that the none of the processes got deadlocked, meaning that sleeps likely work properly.

6 Discussion of Cuncurrency

In the code the structure which is constantly changing through reads and writes is in the buffers, and as such it uses a mutex. This mutex is used whenever the inards of buffer is accessed, this avoids race conditions which could make the program act irregularly. In order to make sure that another process can access a buffer after a process holding a mutex goes to sleep, mutex needs to be unlocked before sleeping. The process tries to reobtain the lock after it has slept to resume what it was going to do. The lock is also always unlocked when an error is encountered so that the processes will be able to keep working even if this one failed.

Since the lock for the buffer is obtained whenever something in the buffer should be changed, no two processes should be able to try to change anything at the same time, which avoids race conditions. The lock is also always unlocked when a change is done or an error is encountered, which makes sure the program avoids deadlocks, which could have come from not unlocking the mutex.

7 Conclussion

Due to shallow testing it may not be adequate to conclude that the program behaves in the exact way it is supposed to do. Some of these features which would need testing are ioctl, failure when given invalid arguments and non-blocking I/O. The program did pass the test which was provided, and could load and unload properly. The the locking of the buffers should work correctly so race condtions should not be encountered. Sleeping should also be working properly since the test ran correctly.

8 Appendix

8.1 Code

```
/* Prototype module for second mandatory DM510 assignment */
#ifndef __KERNEL__
#define __KERNEL__
#endif
#ifndef MODULE
#define MODULE
#endif

#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/wait.h>
/* #include <asm/uaccess.h> */
#include <linux/uaccess.h>
#include <linux/semaphore.h>
/* #include <asm/system.h> */
#include <asm/switch_to.h>
#include <linux/cdev.h>
#include <linux/ioctl.h>

/* Prototypes - this would normally go in a .h file */
static int dm510_open(struct inode *, struct file *);
static int dm510_release(struct inode *, struct file *);
static ssize_t dm510_read(struct file *, char *, size_t, loff_t *);
static ssize_t dm510_write(struct file *, const char *, size_t, loff_
long dm510_ioctl(struct file *filp, unsigned int cmd, unsigned long a

#define DEVICE_NAME "dm510_dev" /* Dev name as it appears in /proc/de
#define MAJOR_NUMBER 254
#define MIN_MINOR_NUMBER 0
```

```
#define MAX_MINOR_NUMBER 1
#define BUFFER_SIZE 1000

#define SET_READ_MAX _IOW('k', 'a', int32_t*)
#define SET_BUFSIZE _IOW('k', 'b', int32_t*)

#define DEVICE_COUNT 2
/* end of what really should have been in a .h file */
struct buffer
{
        wait_queue_head_t inq, outq;
        char *buffer, *end;
        int buffersize;
        char *rp, *wp;
        int nreaders, nwriters;
        struct fasync_struct *async_queue;
        struct mutex mutex;
};

struct dev_driver
{
        struct buffer *readbuf;
        struct buffer *writebuf;
        struct cdev cdev;
};

struct dev_driver *devices;
int buffer_size = BUFFER_SIZE;
int major_number = MAJOR_NUMBER;
int read_limit=0;


/* file operations struct */
static struct file_operations dm510_fops = {
        .owner = THIS_MODULE,
        .read = dm510_read,
        .write = dm510_write,
        .open = dm510_open,
```

```
        .release = dm510_release,
        .unlocked_ioctl = dm510_ioctl};


static void setup_cdev(struct dev_driver *dev, int index){
        int err, devno = MKDEV(major_number, MIN_MINOR_NUMBER + index

        cdev_init(&dev->cdev, &dm510_fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &dm510_fops;

        err = cdev_add(&dev->cdev, devno, 1);

        if (err){
                printk(KERN_NOTICE "Error %d adding device%d", err, i
        }

}

int init_buffer(struct buffer *buf){


        init_waitqueue_head(&(buf->inq));
        init_waitqueue_head(&(buf->outq));
        mutex_init(&(buf->mutex));


        return 0;
}

/* Called when module is unloaded */
void dm510_cleanup_module(void)
{

        /* clean up code belongs here */
        int i;
        dev_t devno = MKDEV(MAJOR_NUMBER, MIN_MINOR_NUMBER);
        if (devices)
```

```
        {
                if (devices[0].readbuf)
                {
                        if (devices[0].readbuf->buffer)
                        {
                                kfree(devices[0].readbuf->buffer);
                        }
                        kfree(devices[0].readbuf);
                }

                if (devices[0].writebuf)
                {
                        if (devices[0].writebuf->buffer)
                        {
                                kfree(devices[0].writebuf->buffer);
                        }
                        kfree(devices[0].writebuf);
                }

                for (i = 0; i < DEVICE_COUNT; i++)
                {
                        cdev_del(&devices[i].cdev);
                }
                kfree(devices);
        }
        unregister_chrdev_region(devno, DEVICE_COUNT);
        printk(KERN_INFO "DM510: Module unloaded.\n");
}

/* called when module is loaded */
int dm510_init_module(void)
{
        dev_t dev;
        int result;
        struct buffer *buf0 = NULL;
        struct buffer *buf1 = NULL;
        /* initialization code belongs here */
        //get dev
```

```
if (MAJOR_NUMBER)
{
        dev = MKDEV(MAJOR_NUMBER, MIN_MINOR_NUMBER);
        result = register_chrdev_region(dev, DEVICE_COUNT, DE
}
else
{
        result = alloc_chrdev_region(&dev, MIN_MINOR_NUMBER,
        major_number = MAJOR(dev);
}
if (result < 0)
{
        printk(KERN_WARNING "device: can't get major %d\n", M
        return result;
}
printk(KERN_INFO "DM510: Hello from your device!\n");

//allocate
devices = kmalloc(DEVICE_COUNT * sizeof(struct dev_driver), G
if (!devices)
{
        result = -ENOMEM;
        dm510_cleanup_module();
        return result;
}
memset(devices, 0, DEVICE_COUNT * sizeof(struct dev_driver));
buf0 = kmalloc(sizeof(struct buffer), GFP_KERNEL);
if (!buf0)
{
        dm510_cleanup_module();
        return -ENOMEM;
}
init_buffer(buf0);
buf1 = kmalloc(sizeof(struct buffer), GFP_KERNEL);
if (!buf1)
{
        kfree(buf0);
        dm510_cleanup_module();
```

```
                return -ENOMEM;
        }
        init_buffer(buf1);

                devices[0].readbuf = buf0;
                devices[0].writebuf = buf1;
                        setup_cdev(&devices[0], 0);

                        devices[1].writebuf = buf0;
                        devices[1].readbuf = buf1;

                        setup_cdev(&devices[1], 1);


        return 0;
}




//needs lock for buffer before call
int allocate_buffer(struct buffer *buf)
{

        buf->buffer = kmalloc(buffer_size, GFP_KERNEL);
        if (!buf->buffer)
        {
                return -ENOMEM;
        }
        buf->buffersize = buffer_size;
        buf->end = buf->buffer + buf->buffersize;
        buf->rp = buf->wp = buf->buffer;
        return 0;
}

/* Called when a process tries to open the device file */
static int dm510_open(struct inode *inode, struct file *filp)
{
```

```
/* device claiming code belongs here */
struct dev_driver *dev;

dev = &devices[iminor(inode)];

filp->private_data = dev;

if (filp->f_mode & FMODE_READ)
{

        if (mutex_lock_interruptible(&(dev->readbuf->mutex)))
        {
                return -ERESTARTSYS;
        }

        if(read_limit){
        while(dev->readbuf->nreaders >= read_limit){
                mutex_unlock(&(dev->readbuf->mutex));
                if (filp->f_flags & O_NONBLOCK)
                {
                        return -EAGAIN;
                }
                if(wait_event_interruptible(dev->readbuf->inq
                        return -ERESTARTSYS;
                }
                if (mutex_lock_interruptible(&dev->readbuf->m
                {
                        return -ERESTARTSYS;
                }
        }
        }


        if (!dev->readbuf->buffer)
        {

                if (allocate_buffer(dev->readbuf) < 0)
```

```
                {
                        mutex_unlock(&dev->readbuf->mutex);
                        return -ENOMEM;
                }
        }

        dev->readbuf->nreaders++;
        mutex_unlock(&dev->readbuf->mutex);
}
if (filp->f_mode & FMODE_WRITE)
{

        if (mutex_lock_interruptible(&dev->writebuf->mutex))
        {
                return -ERESTARTSYS;
        }
        //only allow writer if there is no other writer
        while (dev->writebuf->nwriters > 0)
        {
                mutex_unlock(&dev->writebuf->mutex);
                if (filp->f_flags & O_NONBLOCK)
                {
                        return -EAGAIN;
                }
                if (wait_event_interruptible(dev->writebuf->o
                {
                        return -ERESTARTSYS;
                }
                if (mutex_lock_interruptible(&dev->writebuf->
                {
                        return -ERESTARTSYS;
                }
        }
        if (!dev->writebuf->buffer)
        {
                if (allocate_buffer(dev->writebuf) < 0)
                {
                        mutex_unlock(&dev->writebuf->mutex);
```

```
                                return -ENOMEM;
                        }
                }
        dev->writebuf->nwriters++;
        mutex_unlock(&dev->writebuf->mutex);
        }

//maybe nonseekable_open?
return 0;
}



/* Called when a process closes the device file. */
static int dm510_release(struct inode *inode, struct file *filp)
{

        /* device release code belongs here */
        struct dev_driver *dev = filp->private_data;
        //maybe fasync?
        if (filp->f_mode & FMODE_READ)
        {
                if (mutex_lock_interruptible(&dev->readbuf->mutex))
                {
                        return -ERESTARTSYS;
                }
                dev->readbuf->nreaders--;
                if (dev->readbuf->nreaders + dev->readbuf->nwriters =
                {
                        kfree(dev->readbuf->buffer);
                        dev->readbuf->buffer = NULL;
                }
                mutex_unlock(&dev->readbuf->mutex);
                wake_up_interruptible(&dev->readbuf->inq);
        }
        if (filp->f_mode & FMODE_WRITE)
        {
                if (mutex_lock_interruptible(&dev->writebuf->mutex))
```

```
                {
                        return -ERESTARTSYS;
                }
                dev->writebuf->nwriters--;
                if (dev->writebuf->nreaders + dev->writebuf->nwriters
                {
                        kfree(dev->writebuf->buffer);
                        dev->writebuf->buffer = NULL;
                }
                mutex_unlock(&dev->writebuf->mutex);
                wake_up_interruptible(&dev->writebuf->outq);

        }
        return 0;
}

/* Called when a process, which already opened the dev file, attempts
static ssize_t dm510_read(struct file *filp,
                                                char *buf,
/* The buffer to fill with data     */
                                                size_t count,
/* The max number of bytes to read  */
                                                loff_t *f_pos) /* T
*/
{

        /* read code belongs here */
        struct dev_driver *dev = filp->private_data;

        if (mutex_lock_interruptible(&dev->readbuf->mutex))
        {
                return -ERESTARTSYS;
        }
        while (dev->readbuf->rp == dev->readbuf->wp)
        {
                mutex_unlock(&dev->readbuf->mutex);
                if (filp->f_flags & O_NONBLOCK)
                {
```

```
                return -EAGAIN;
            }
            //pdebug??
            if (wait_event_interruptible(dev->readbuf->inq, (dev-
            {
                return -ERESTARTSYS;
            }

            if (mutex_lock_interruptible(&dev->readbuf->mutex))
            {
                return -ERESTARTSYS;
            }
        }

        if (dev->readbuf->wp > dev->readbuf->rp)
        {
            count = min(count, (size_t)(dev->readbuf->wp - dev->r
        }
        else
        {
            count = min(count, (size_t)(dev->readbuf->end - dev->
        }
        if (copy_to_user(buf, dev->readbuf->rp, count))
        {
            mutex_unlock(&dev->readbuf->mutex);
            return -EFAULT;
        }
        dev->readbuf->rp += count;
        if (dev->readbuf->rp == dev->readbuf->end)
        {
            dev->readbuf->rp = dev->readbuf->buffer;
        }
        mutex_unlock(&dev->readbuf->mutex);
        wake_up_interruptible(&dev->readbuf->outq);
        //pdebug??
        return count;
}
```

```
// returns how much space is empty of the buffer, returning -1 if emp
static int spacefree(struct buffer *buf)
{
        if (buf->rp == buf->wp)
        {
                return buf->buffersize -1;
        }
        return ((buf->rp + buf->buffersize - buf->wp) % buf->buffersi
}

static int getwritespace(struct dev_driver *dev, struct file *filp)
{
        while (spacefree(dev->writebuf) == 0)
        {
                mutex_unlock(&dev->writebuf->mutex);
                if (filp->f_flags & O_NONBLOCK)
                {
                        return -EAGAIN;
                }
                if (wait_event_interruptible(dev->writebuf->outq, (sp
                {
                        return -ERESTARTSYS;
                }
                if (mutex_lock_interruptible(&dev->writebuf->mutex))
                {
                        return -ERESTARTSYS;
                }
        }
        return 0;
}


/* Called when a process writes to dev file */
static ssize_t dm510_write(struct file *filp,
                                                const char *buf, /
*/
                                                size_t count,
```

```
                                                        loff_t *f_pos)
/* The offset in the file           */
{

        /* write code belongs here */
        struct dev_driver *dev = filp->private_data;
        int result;

        if (mutex_lock_interruptible(&dev->writebuf->mutex))
        {
                return -ERESTARTSYS;
        }

        result = getwritespace(dev, filp);
        if (result)
        {
                return result;
        }

        count = min(count, (size_t)(spacefree(dev->writebuf)));
        if (dev->writebuf->wp >= dev->writebuf->rp)
        {
                count = min(count, (size_t)(dev->writebuf->end - dev-
        }
        else
        {
                count = min(count, (size_t)(dev->writebuf->rp - dev->
        }
        if (copy_from_user(dev->writebuf->wp, buf, count))
        {
                mutex_unlock(&dev->writebuf->mutex);
                return -EFAULT;
        }
        dev->writebuf->wp += count;

        if (dev->writebuf->wp == dev->writebuf->end)
        {
                dev->writebuf->wp = dev->writebuf->buffer;
```

```
        }

        mutex_unlock(&dev->writebuf->mutex);

        wake_up_interruptible(&dev->writebuf->inq);

        return count; //return number of bytes written
}



/* called by system call icotl */
long dm510_ioctl(
        struct file *filp,
        unsigned int cmd,  /* command passed from the user */
        unsigned long arg) /* argument of the command */
{
        /* ioctl code belongs here */
        printk(KERN_INFO "DM510: ioctl called.\n");
        switch(cmd){
                case SET_READ_MAX:
                        if (arg < 0){
                                return -EINVAL;
                        }
                        read_limit = arg;
                        break;
                case SET_BUFSIZE:
                        if (arg < 0){
                                return -EINVAL;
                        }
                        buffer_size = arg;
                        break;

                default:
                        return -ENOTTY;
        }
        return 0; //has to be changed
}
```

```
module_init(dm510_init_module);
module_exit(dm510_cleanup_module);

MODULE_AUTHOR("Sebastian Larsen");
MODULE_LICENSE("GPL");
```