Assignment2 OS device drivers

Kristian Degn Abrahamsen, krabr21

April 2023

Contents

Introduction

The code was developped by me and Philip Jos Rosenlund Andersen
In this assignment we are to implement a device driver, which involves two bounded buffers, which are to be written to and read from by two devices. Device drivers are usually implemented as an interface between some hardware and the operating system. When you then want to use some hardware, you load its module/device drivers, and it can then be used. However as we in this course focus more on the software side, we implement only a software solution. Therefore the implementation of this device driver is focused on manipulating a shared array. This then includes having concurrency, device driver manipulation and CPU-optimazation in mind. The implemented device driver involves: One device writes to *buffer0* and reads from *buffer1*, and the other is vice versa.

1 Design Choises

1.1 dm510Q_dev & dm510_buffer

Each device has its own *struct dm510_dev*, this is such that each specific device can be accessed via the *struct cdev*-attribute. This combined with the *container_of()*-method, each device structure can be accessed when using each method in the device. This holds crucial information, such that the methods can get access to all information it needs. The *struct dm510_buffer* is a structure that is created such that it is easier to access the information for which buffer each device needs to write or read from.

1.2 Sleeping

As these buffers can be accessed concurrently, it is sometimes better to put processes to sleep, if there fx is no data yet to be read. Therefore when processes tries to read, and there is no data yet to be read, they are put to sleep. Vice versa when the buffer is full, the writing process needs to be put asleep. If the processes indicates they do not want to be blocked, the methods will return immediatly, to not violate this.

1.3 Concurrency

These devices can be accessed by multiple processes at the same time, therefore concurrency needs to be put in mind. There are multiple ways to implement concurrency measures, for ensuring mutual exclusion, however in this implementation, the easiest locking-procedure is used: a single lock for reading and writing, such that you must hold the lock before doing either and if you want to update the shared structures. This locking strategy is used, to ensure a simpler implementation.

1.4 Allowed methods

The implementation of this module is done via *dm510_init_module()* which initializes the module, when you want to load it. Of course an initialization method is important to implement. *dm510_cleanup_module()* is called when cleaned up, ensuring everything done in the *open()* and initializitation is undone. *open()* This makes it possible to actually use the device, opening the specific device such that it can be used. *release()* is implemented such that each opened device can again be closed from the file descriptor given in *open()*. *read()* and *write()* needs to be implemented such that we can actually write and read the buffers. And lastly *ioctl()* is chosen to be implemented aswel, such that we can control the device drivers number of allowed readers and the buffer sizes at runtime.

1.5 Helper attributes

The device driver has two helper attributes to it, which is a pointer to the created devices, accessed like it is an array, and also an *int* to keep track of the amount of allowed readers.

2 Implementation

2.1 dm510_dev & dm510_buffer

The dm510_dev looks like this

```
1 struct dm510_dev
2 {
3     int minor_number;
4     struct cdev cdev;
5     wait_queue_head_t read_q, write_q, open_q;
6     struct mutex mutex; // lock for opening and closing the pipes,
      and updating this dev.
7     char *wp, *rp;
8     int nreaders, nwriters;
9 };
```

The minor number is used to knowing which buffer we want to access as well as which device is being accessed, which done a lot throughout the implementation of the methods that are implemented. The **struct cdev cdev** is used to setup the character device with the methods *cdev_add()* and *cdev_init()*, which are defined in the *¡linux/cdev.h¿* headerfile. The *wait-ques* are used as read queue, write queue and open queue for the device. These will be discussed later in the sleeping subchapter. Thhe **mutex** is used when wanting to access or modify the device, to ensure mutual exclusion when updating this. The write and read pointers are used to point to the specific buffer for which device should read and write from. *nreaders* and *nwriters* are used to keep track of details sorrounding the how many readers and writers are currently in this device.

2.2 Sleeping

As discussed, when a process wants to read from the device, but no data is available, the process is put to sleep if it allows blocking. This is checked

```
    if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
```

Where the *filp* is the filer pointer given when opening the device. If the process is allowed to sleep it is put into the corresponding *wait_queue*. The method used for this is

```
    wait_event_interruptible(wait_queue_head_t *queue, condition)
```

This will put the process into an interruptible sleep, which can then be woken up by the corresponding function

```
    wake_up_interruptible(wait_queue_head_t *queue)
```

This will wake the waiting processes up which are in the sleeping queue, but the processes are only signaled, and they will only wake up if the corresponding condition evaluates to true.

Now it is important notice that before any write or read should go to sleep it shall release any lock it is holding, and require it afterwards again. The just woken process should also make sure that any dependancies it has, which was checked before the sleep, is also what it should be (else it has to update it), as it does not know how long it has slept for, and it cannot be sure of any state but what is local to the process.

Anoter sleeping mechanism used is the

```
    DEFINE_WAIT(wait);
    prepare_to_wait(&queue, &__wait, state);

    schedule();
    finish_wait(&queue, &__wait);
```

Which is just using different sleeping methods, to acheieve the same thing: sleeping.

2.3 dm510_fops

```
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
};
```

Is used to when the *cdev* is initialized, to register which function in the device driver should be used when the functions are called to a given device.

2.4 Initialization

The registration of the cdev, as discussed, is done via

```
1    cdev_init(&dev->cdev, &dm510_fops);
2    cdev_add(&dev->cdev, devno, 1);
```

which adds a device to the specifc *devno* which is given by

```
1    devno = MKDEV(MAJOR_NUMBER, index);
```

which hooks up the device minor number to the *MAJOR_NUMBER* with a given index/minor-number, stored in a *dev_t*. Before we actually can hook up any devices to a device driver, we first need to allocate a range of device drivers using the method

```
1    register_chrdev_region(dev_t from, unsigned int count, const
     char *name)
```

Where the name is the device or driver name, from is the first nubmer in the range and count is how many additional numbers from the *from*.

Before initializing the *cdevs* for the two structures, we allocate the space for two *struct dm510_dev* and two buffers using

```
1     devices = kmalloc(DEVICE_COUNT * sizeof(struct dm510_dev),
     GFP_KERNEL);
2
3     buffers = kzalloc(2 * sizeof(struct dm510_buffer), GFP_KERNEL)
     ;
4
```

Next The buffer attributes are initialized, allocating a char array corresponding to the size of the defined buffer size, pointing the head to it, and the end attribute to the end of the buffer. The two *dm510_dev* devs are similarly initialized, by initializing the cdevs using the described methods, the wait-que and mutexes are intiialized using

```
1    init_waitqueue_head(wait_queue_head_t *queue)
2    mutex_init(mutex *mutex)
```

the less interesting ones are not descibed, but the attributes are initialized.

2.5 Cleanup

This is called upon unloading the devices. Firstly if the devices are not initialized we just return, as there is nothing to do if so. Else we delete the cdevs using

```
1    cdev_del(cdev *cdev)
```

Then the buffers and devices are freed, and lastly the device regions are unregisterd, i.e the minor numbers and major number associated with the driver is unregisterd using

```
1    unregister_chrdev_region(dev_t from, unsigned int count)
```

2.6 Open

The open method is used to open a specific device and get a file descriptor for this device. It uses the *container_of()* method to grap the cdev for the device, checks if we are able to enter, as if we want to open the file for reading, we may have reached the limit of the number of readers. If it has to wait for a reader to leave the device, it releases the semaphore and put it self to sleep using the *open-q* for the device, and the condition that the number of readers for the device drops below the number of readers that are allowed. If however it may not be blocked, the method returns immediatly. Lastly, when woken up it acquires the lock again, and increment the number of readers if the file is opened for reading, and writers if the file is opened to write and of course it releases the lock.

2.7 Release

Will grap the lock for the device, and decrement number of readers and writers if the file was opened for reading and or writing. It then wakes up the waiting processes in the *open-q*, as it may just have decremented the number of readers. And lastly release the lock, and return 0 signaling the operation went well.

Read the read-method gets a hold of the lock, and checks if there is something to read. If not, it releases the lock, and goes sleep on the condition that something becomes available, i.e the write pointer and the read pointer is not the same in the buffer it is pointed to. When woken up it acquires the lock again, and reads up until the write pointer if it ahead of the read-pointer, or up until the end of the buffer. It then copies the content to the user-space buffer, using the method *copy_to_user()*. We then increment the read pointer, if it is at the end, it wraps around to the start of the buffer, else it just incremented corresponding to the amount of bytes we have read. Lastly it wakes up the readers, and releases the lock, while returning the count of how many bytes read.

Write The write method starts by checking how much space is free in the buffer, i.e checking where the write-pointer is corresponding to the read-pointer. If this is 0, we go to sleep on the buffers write-queue, waiting for a read call to give more buffer-space. Before sleeping it of course unlocks the lock and after the sleep it acquires it again. Then it copies the specified number of bytes into the buffer, or up until there is no more space in the buffer. The write-pointer is then incremented matching the number of bytes copied into the buffer, releases the lock, wakes up readers on the buffers read-q, and return the amount of bytes copied into the buffer.

Ioctl The ioctl-method is used to update the buffer-size and the number of readers at runtime. This is done using the convention of having a magic a number, which is chosen to be 'N' and a sequential number for the specific command. As the *ioct-number.rst* tells us that for the magic number 'N', there are free sequential numbers from *80-83* in hexadecimal, matching *128-131* in decimal. The command for reading the buffer size is 'N' 128, reading number

of readers 'N' 129 ... and so on. The methods *__put_user()* and *__get_user()* are used to make small copies from the given argument to the method. These can be used as we have already used *access_ok()* to check if we may access the user-space pointer. If we may not we simply return *-EFAULT* signaling a bad address was provided. The changing of the buffer-size may only be used before starting to writing or reading to the buffer, as it will update the write and read pointers to a new allocated char buffer, so any previous changing to these will be overwritten, and data will be lost.

3 Concurrency

If the program is used in a concurrent fashion, nothing will break as we use semaphores, and as the locking of these semaphores are done in an atomic fashion, multiple processes cannot get into the mutual exclusion area at the same time, since it is protected by this semaphore.

4 Tests

4.1 test_ioctl

This tests all the ioctl commands, testing to see if the buffer size has been changed, and that we can read it again. The same thing is done for the number of readers.

4.2 test_read_write

Is written by Daniel, and we can see that we get the expected and actual to match for both read and write.

4.3 additional tests

The procedure of implementation of sleeping when a process tries to open one of the devices, while there are a maximum number of readers, could also be tested. However as it uses the same locking and sleeping as the read and write methods, it is trivial.

5 Conclusion

In this assignment I have implemented a device driver, that has two devices, which can be used as two bounded buffers for reading and writing. These can be opened and used by any program. They can be loaded and unloaded, and used by the programs as pipes to talk to each other. Locking and sleeping is used for concurrency-measures. The module is writting in kernel-space, and uses many implementations of already written methods, which makes the task incredibly easy to implement.