Kernel Module - Project

Bavith Kirshnakumar [Bakir23],
Emil Vadgaard Nielsen [Emiln17]

April 2025

Contents

1 Introduction

The goal of this project is to implement a kernel module, which creates and implements two working character devices, "dm510-0" & "dm510-1". Sending data/producing data to one character device, will allow the retrieval/consumption of the data via the other character device.

2 Design Decisions

In choosing the design and behaviour of our kernel module, we chose to make one major decision. We wanted our data to be persistent, such that data is preserved until the module is unloaded. Other design concepts include how we use blocking and non-blocking I/O, how we connect the two devices to achieve reading to one device, and retrieving that data from the other device, and how we handle errors. In this section these things will be explained in further detail.

2.1 Connection between the two devices

We have our two devices named dm510-0 and dm510-1. Our task was to make sure, that when a message is written into device dm510-0, that message is retrieved by reading from device dm510-1. This we simply achieved by a simple trick, in which when a user requests to write a message in one device, it is in reality just written into the other. This means that each device stores a reference to the other device, and we access the buffer of the opposite device, and its information.

2.2 Data persistence

As mentioned, we chose to keep the data persistent as long as the module is loaded. This enables us to write a message into a device, and when said device gets released, the data inside, is preserved. When opened once again, the message can still be retrieved.
Another option is when a device is released, the read, write and buffer get reset, and the memory gets freed. This means that each time a device is opened, memory is allocated for the buffer and the rest of the pipe-struct gets initialized. However, we chose to allocate memory when the module has been loaded, and everything has been initialized there. We only then, free the memory when the module gets unloaded, and this enables us to keep the data when a device is released. One could argue, that this approach takes up more space in memory, which is true, however, this is necessary to keep the data. One workaround could be to delay initialization for when the device is opened the first time, and only do it the first

time it gets called. This would then save memory in the case that the module has already been loaded, but no devices have been in use, before being unloaded again. However, we would argue, it is a safe assumption, that if the module is loaded, the user also intends to use it, and as such, there is no need to delay initialization.

2.3 Error handling

The main error handling for the kernel module are:

1. ERESTARTSYS: If our blocking/Non-blocking I/O failed to execute during any function call, ERESTARTSYS will be returned.

2. ENOMEM: If memory allocation is not possible for the buffer, ENOMEM will be returned. Occurs during the initialisation of devices and changing the size of buffer, from *kmalloc*.

3. EBUSY: If number of readers exceed the maximum allowed number of readers, EBUSY will be returned to indicate that the device is busy.

4. EFAULT: During writing, reading, getting buffersize and getting max readers, if *copy_to_user* & *copy_from_user* fails to execute, EFAULT will be returned. This is mostly due to the arguments entered into the functions, are not formatted in the correct way, or the address being wrong.

2.4 Concurrency

To allow several processes to use the character devices at the same time, we implement a mutex lock in both pipes, and include mutex blocks which use these locks. The mutex lock is used in functions to disallow race condition, such as in the cases of:

1. Reading data from a device: The device first locks the mutex, to avoid race condition with any other processes who are trying to read at the same, and to avoid writers writing into the buffer, while we are trying to read. If there is data to read, we then skip the while-loop, and then execute the necessary things to be able to read the message from the device. Afterwards, we open the mutex, to allow another reader/writer to continue their execution.
   If there is no data to read, we enter our while-loop, we open the mutex lock, as we now have to wait for writers to write data to the device. Once we have been awakened from our non-blocking I/O by a writer, we lock the mutex, and check whether we can continue our execution of the rest of the function.

2. Writing data into a device: The device first locks the mutex, and checks whether there is space or not to write inside the buffer. If there is no space, we then release our lock, to allow readers to read from the buffer, to free up space. Once we have been awakened from our non-blocking I/O by a reader, we lock the mutex, as we can now continue our execution for writing into the buffer.

Additionally, to avoid a process busy-waiting and creating too much overhead, we then put the process to sleep.
For read, if there are no data in the buffer, we then go to sleep, to eventually be woken by a writer, as this means that we now have data inside the buffer. Once we have read, we then awaken any writers, who have been waiting for more space in the buffer, to be able to write into it.
The same applies for write, as it goes to sleep if the buffer is full. Once we have been awoken by the readers, indicating there is now new space, we then write into the buffer, and awaken any readers who have been waiting on data.

3 Implementation - Kernel Module

Here we will discuss in more detail how our kernel module was created. We will show our data structures, how we initialize and clean up our devices, how we open and close a device, and how we write into and read from a device. We also explain our I/O control, and how we can change internal parameters of the devices.

3.1 Datastructures

Our device data structure is a normal pipe. However, as mentioned, when a message is put on one pipe, it is actually physically placed in the other device's pipe. This means we keep a pointer to the location of the other pipe.

```
1 struct DM510_pipe
2   wait_queue_head_t inq, outq;
3   char *buffer, *end;
4   int buffersize;
5   char *rp, *wp;
6   int nreaders, nwriters;
7   struct mutex mutex;
8   struct cdev cdev;
9   struct DM510_pipe *other_pipe;
```

Listing 1: Pipe structure for devices

As shown above, our pipe contains two queues, *in* & *out*, for processes trying to read or write from the device. We have our buffer, which is made up of a pointer to the start of the buffer, and one to the end of the buffer. Buffersize is included, to avoid calculating the size, for each use case. For our read and write pointers, we have two integers to indicate how many readers and writers are in queue. We have a mutex structure, which enables us to lock critical sections of code, and we can do this both with block and non-blocking I/O. We then have our character device structure, and lastly a pointer to the other pipe as mentioned.

3.2 Initialization and Cleanup

Our *init* and *cleanup* functions are run when the module is loaded and unloaded, respectively. They are mainly involved in setting things up, allocating space in memory, and setting up different structures from libraries. Our *dm510_init_module* sets up the devices with a major and minor number. It then registers a region for our character device, such that our two character devices now show up in our device listings. We then allocate space in memory for both of the pipe data structures, and initialize the memory to 0, such that no lingering abandoned data can cause problems. We then initialize the pipes in their entirety, as mentioned earlier. We initialize both queues, the mutexes, and then initialize the character device in a helper function called *DM510_cdev_setup*, in which we set the owner to our module, and add the device. We then allocate memory for our buffer, set all the pointers and the buffersize. Lastly, we set the pointer to point to the other pipe, such that we can access the opposite pipe. If errors occur in allocation or any setup process, we return the relevant ERRNO code, and if needed, unregister the character device and delete it.

When the module is unloaded with *dm510_cleanup_module*, we then delete the character device and free the buffer for each pipe. We then free the pipes and unregister the character devices region.

3.3 Open and Release

When we want to open a device, the first step is to retrieve our specific pipe-structure from the device we are requesting to open. We achieve this with the *container_of* function, which retrieves the specific pipe struct, from the device. We then save the struct in the file handler we got from user-space. This is the basic preparation for opening our device. We then acquire the lock, such that we can increment the number of readers and writers, and then give back the lock. Finally, we return the function *nonseekable_open*, as to mark that our pipe is non-seekable.

In a much similar fashion, when a device is released, we first lock the device, then decrement the readers and writers, unlock the device, and return.

3.4 Read

When calling the read function, one calls with the device to be read from, the buffer to be filled with data, and a maximal integer amount of bytes to be read. Then we acquire the lock. If the number of readers is not 0, and we exceed the maximum number of readers, we return the error code $EBUSY$, indicating that the queue is filled. We then enter a while loop if the read pointer and write pointer are equal, indicating that the buffer is empty, and there is nothing to read at the moment. This means that we wait until some data is written into the buffer. In this while loop, we first give up the lock, and then enter the wait queue, which means we now wait to be woken up. When woken up, we immediately grab the lock for the device, and we should be able to leave the while loop, as a writing process should have woken us up.
Now we actually begin reading; however, there are two cases for reading. Either the write pointer is in front of the read pointer, or it is behind us. In the case it is behind, that means the write pointer has wrapped around the buffer, and has begun writing to the beginning of the buffer. In either case, we find how much data to send back. We do this by setting the number of bytes to read, to be the minimum of either the amount requested, or the amount left to be read from the buffer. This we copy back to the user. If one requested to read more than to the end of the buffer, and there is in fact more to read from the beginning of the buffer, the user is responsible for fetching the rest of the data, by calling the read function once again.
When reading is done, and sent to user space, the read pointer is set to the correct position, and the lock is released. Then we wake up any processes in queue to write, if the buffer was full.

```
1 if (dev->wp > dev->rp)
2     count = min(count, (size_t)(dev->wp - dev->rp));
3   else
4     count = min(count, (size_t)(dev->end - dev->rp));
5   if (copy_to_user(ret_buf, dev->rp, count)) {
6     mutex_unlock (&dev->mutex);
7     return -EFAULT;
8   }
```

Listing 2: Calculation of how much data is returned, in case of wrap or non-wrap. Additionally, error handling if message was not presented to user-space

3.5 Write

The write function is the most extensive of all the functions, and we have made two helper functions. One to check how much free space is available, and one to put writing processes into queue, if the buffer is full. The *spacefree* function, calculates the the amount of bytes left in the buffer, and returns it as an integer. However, we subtract the amount of space by one, so as to not accidentally place the write pointer on top of the read pointer, because then we would no longer be able to tell whether the buffer is full or empty.
The *dm_getwritespace* function, wants to gain access to writing, as for each device, only one process can write at a time. This function mirrors a part of the read function, wherein, we had a while loop, to put the process to sleep while the buffer was empty. This is exactly the same for the write function when the buffer is full. First, we release the lock, given that the buffer was full. Then we check if the device was opened in non-blocking mode, and if so return the ERRNO code EAGAIN. Then we place the device into queue, where we await being woken up, by a reading process, which empties out some of the buffer. When we are awoken, we immediatly grab the lock, and return to the write function.

Now we move onto the write function itself. When we call the write function, we give the parameters: The device in which we wish to write in, a buffer containing some data, and an amount to be written from the given input buffer. The key to the write function is that when we get the requested device to write in, we instead access the pointer to the opposite device, and write into this device's buffer instead. We then grab the lock and call the *dm_getwritespace* function explained above. Then we get to actually writing in the buffer. This function works as much as the two cases in the read function, where we have the normal case, where we write to the end of the buffer, or we write up to the 'read pointer − 1'. However, before this, we also determine if there is enough space to write the entire message using the $spacefree$ function. Then we write into the buffer, update the write pointer, and check if we should wrap the write pointer to the beginning of the buffer. As with the read function. If only part of the message was written, the user is responsible for writing the rest. The function also returns the amount of bytes actually written, and the user can then figure out how much they need to write, to have written all of the message.

3.6 IOCTL

When making our I/O controller, we were tasked with adding functionality to set the buffer size, and set a max capacity for how many readers may be in the queue at a time. Additionally, we added the ability to read the current buffer size and the current maximum capacity for readers. Outside of the function, we define our functions: *DM_IOCSBUFFER*

(To set buffer size), *DM_IOCGBUFFER* (To get buffer size), *DM_IOCSMAXREAD* (To set the max_readers) and *DM_IOCGMAXREAD* (To get the max #readers), and we define a magic number to be $'e'$.

When *ioctl* is called from user-space, we expect to be given the parameters: The device, a function to execute, and some data or a buffer for data to be placed in. No matter what function we call, we first retrieve the pipe structure from the given device. Then, if *ioctl* is called with the *DM_IOCSBUFFER* function, we use a helper function called *set_buffer*. In this function, we copy the new size from user space to kernel space, then we acquire the lock from the given device, free the buffer, and allocate a new buffer with the new size. We then reinitialize the different pointers in the pipe. This also means that the data, from before *ioctl* was called, is lost. If the buffer size is increased, it could be possible to preserve the data; however, for simplicity's sake, we chose to lose the data. If *ioctl* is called with the function to get the buffer size, we simply copy to user-space, from the given buffer, with the size of the buffer.
This I/O control call, enables us to have two different buffer sizes for each device.

The *DM_IOCSMAXREAD* function, works almost the same as the *DM_IOCSSETBUFFER* function, except the max capacity for the readers is a global parameter, meaning every device has the same capacity. This could also have been device specific, but we decided to keep it global. When the max capacity is set to zero, this means that there can be infinite readers waiting.

4 Tests

4.1 Description & Motivation

When testing our module, we made five tests, to test different aspects of our module. These tests serve to show capabilities of our module, and show both functionality but also limitations of our module. Below is a list of our tests, and later we will explain each in further detail. In this list, are also timestamps, to where in the video the programs are executed.

1. moduletest.c (0:13)
2. full_buffer.c (0:31)
3. read_empty.c (0:44)

4. buffersize.c (0:57)

5. maxReaders.c (1:06)

4.1.1 moduletest.c

This test program makes two processes which each opens their own device; $dm510$-0 and $dm510$-1 respectively. There are two helper functions: *read_all* and *write_all*, which actually writes into the devices and ensures that all bytes are written, and that the data is not lost. One of the processes then writes an increasing value into the device. The value is an integer, which in Linux is commonly 4 bytes. Then the process also calculates the expected value that has been written to its device. Then the process reads all the input the other process wrote, and calculates both the expected and actual value for the sum. The other process does the same, but reads first, then writes, which means the value is already 10.000 before it begins writing.

4.1.2 full_buffer.c

This test program fills our buffer and shows what is read. Here we create 2 processes. One process sleeps, which we will return to later. The other process writes "Hello World" to $dm510 - 0$ device. Then it tries to write the numbers 0 through 9, but the buffer is full, so it is placed in a wait queue. When the sleeping process awakens, it reads, which makes space in the buffer. However, now the read and write pointers are right besides each other, so it can only manage to write a single byte/character, which is '0'. We then write the same numbers again, and manage to write a lot more, and we read the numbers 0 through 7. There is also some garbage data at the end. This is not as a consequence of the device buffer, but instead from the output buffer. The print function prints until it finds a '\0' terminator, which it won't find, and instead prints whatever left over data is in the output buffer.

4.1.3 read_empty.c

This program tests what happens when one tries to read an empty buffer. This program makes two processes. One sleeps immediately, the other tries to read, but is placed in a wait queue. The other process then awakens and writes into the device, which wakes up the reading process from the queue, and it can then read the message.

4.1.4 buffersize.c

This test simply shows the ioctl *DM_IOCSBUFFER* function works, and also demonstrates that 2 devices can have different sizes.

4.1.5 maxReaders.c

This program shows that the *DM_IOCSMAXREAD* function works. Before the call, the max readers is set to zero, which means that there can be an infinite number of readers in the queue. After the call, max-readers is set to two. We then create three processes, each of which tries to read. Since max readers is set to two, one of these processes is bound to fail, however, it will be different every time which one fails.

5 Conclusion

With the implementation of the kernel module, we have now created 2 functional character devices for user-space. Writing to one device allows it to be read from another device. We have ensured that both read and write run smoothly and can be run in concurrent scenarios, and additionally allowed the changing of the buffer size and number of readers.