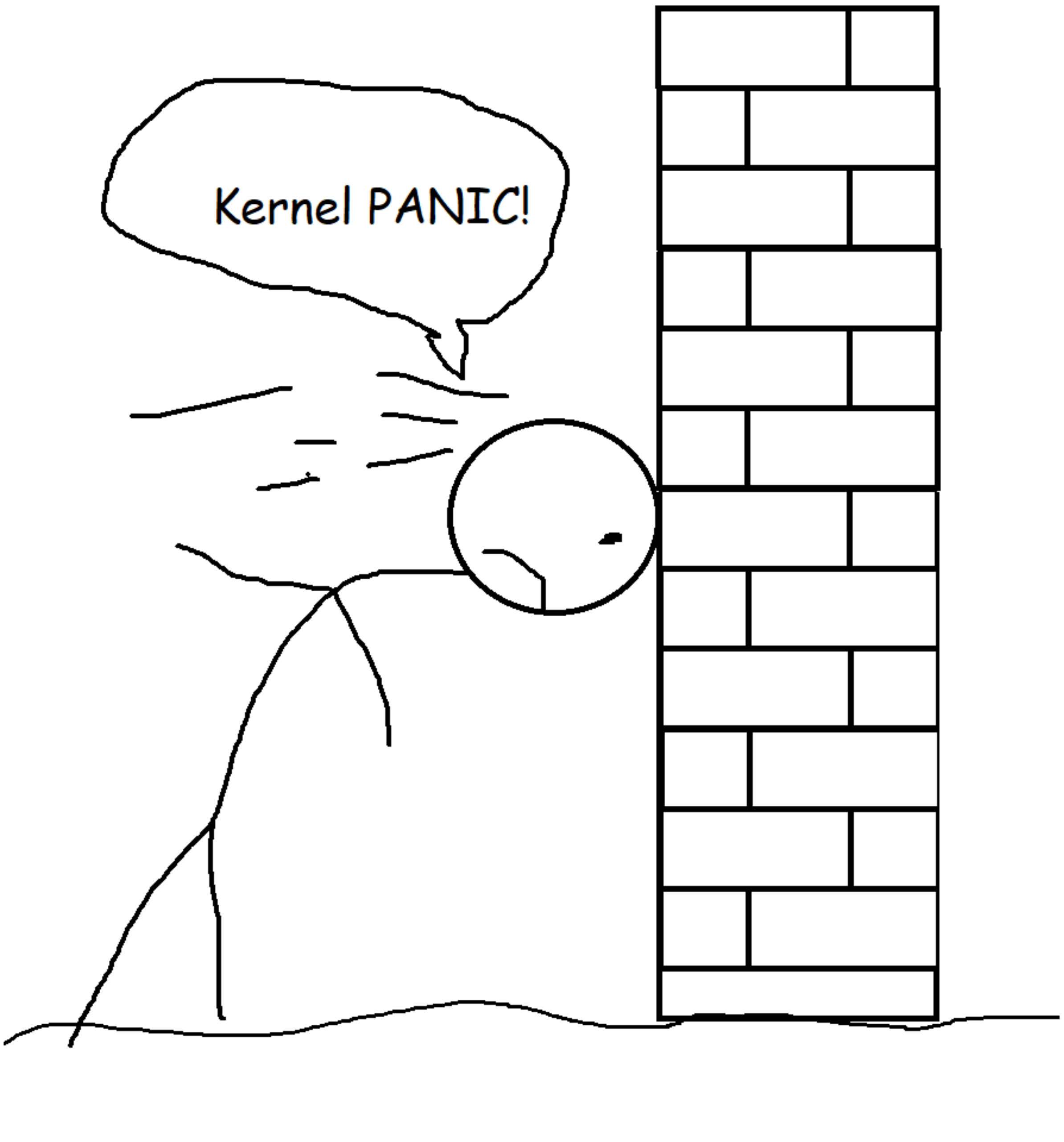
Assignment 2

Kernel Module



Danny Nicolai Larsen, in cooperation with: Mikkel Brix Nielsen, Steffen Bach

DM510

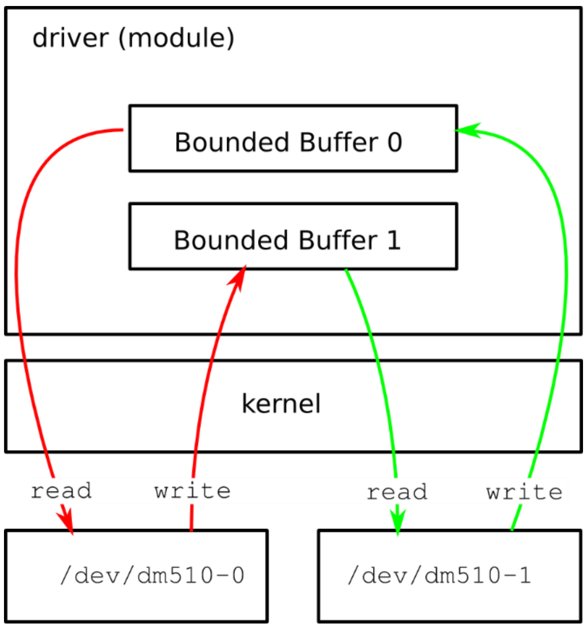
Indhold

Introduction 2
Design 3
Constants 3
DM510 datatype 3
Load / unload 3
Implementation 4
Load / unload 4
Open / close 5
Read / write 5
ioctl 6
Testing 6
Concurrency 8
Conclusion 8

Introduction

For this assignment, the task was to make a kernel module which implements a driver that makes it possible for the user to interact with two character devices, which in turn needs to be able to communicate with each other.

In contrast to a normal device driver, which acts as an intermediary between hardware and the kernel and makes the hardware work in the operating system, the device driver will act as an intermediary between two processes, which will communicate through the device driver.



The driver must work as illustrated in the above image, where both processes can be producers and consumers, thus when a process writes to one character device, the data is stored in the other character devices' buffer, and vice versa. If the buffer is full, the process must wait until another process has read from it, and if a buffer is empty when a process tries to read from it, the same thing applies, and that process will have to wait until something is written.

Several processes should be able to read from the same device at the same time, but only one process should be allowed to write to each device concurrently, which should be achieved by use of some sort of blocking mechanism.

Device control via the oictl system call should be implemented, and as a minimum could adjust buffer size and setting the max number of reading processes.

This module must of course be implemented for user-mode Linux,

Design

This driver has been heavily based on the scull driver from the book, Linux Device Drivers 3rd edition, with the biggest changes being the change from having one char driver to having two that interacts with each other and slimmed down to only use the parts needed for this assignment. Thusly, much of the design has been chosen based on the code given, only changing what is needed. The considerations were as follows.

Constants

There were several constants that, for readability and maintainability, needed to be defined. It made sense to define the chosen major and minor numbers along with the number of readers and writers, buffer size and device count that the driver should have as standard, since these might be changed later, and having these as constants makes it easier to maintain, since it would then only have to be changed in one place. The rest of the constants chosen are related to the ioctl and these also make sense to keep as constants as it makes the ioctl implementation more readable, and it's easy to see which ioctl methods are implemented.

DM510 datatype

For the character drivers, a struct DM510 was needed. This struct contains several parameters which were needed for the chosen implementation. Firstly, two queues were needed, one for writers and one for readers to wait in, if the device was busy.

Since each character driver would have a buffer to write in, the struct should also have a pointer to the starting point as well as the end of this buffer, as well as a read and write pointer to know if there is anything in the buffer to read, and to know if there's any space to write.

As IO blocking should be implemented based on the number of readers and writers, the struct should also have two counters keeping track of these, and for each separate char driver, since even though only one writer could write to a device at the same time, it should still be possible to write to the other device at the same time. To avoid race conditions on these parameters and to ensure that no processes modify the devices at the same time, each character driver also has a mutex lock.

Lastly, the struct also has a cdev structure, which is the actual char device structure belonging to the char driver.

Load / unload

When the module is loaded, the char drivers should be initialized, memory for them should be allocated and their queues initialized, then the char devices should be initialized as well. To follow the scull driver, it was chosen to register the device numbers, instead of having them dynamically assigned, which would have been better if this was a real-world setting, as it more reliably avoids conflict with other devices.

When the module is unloaded, it should delete the devices, free the memory for the buffers and the driver structs, and unregister the device numbers (major and minors)

When a device file is opened, it should grab the mutex for both devices, since even when writing to one device, access to the buffer on the other device is needed. It should then check what intent the call has, if it needs to read or write, and then check if the max number of readers or writers has been reached and return an error in that case. It then needs to increment the corresponding counter, unlock the mutex and return the opened file.

When a device file is closed, it should acquire the mutex for the relevant file, decrement the relevant counter, free the buffer, and unlock the mutex.

When a process attempts to read an opened file, it should grab a mutex to make sure the written content isn't modified while reading, then read as much data as is available, and pass it to a user space buffer. This driver implements a circular buffer, meaning, if the buffer reaches the end, it starts over from the beginning, if not blocked by the write-pointer, however, if the process reads, and reaches the end-pointer it will only print until end, put the read-pointer at the start of the buffer, and leave the rest of the read for the next cycle.

However, if there is nothing to read in the buffer, it should put itself in the reading queue, and go to sleep until there is something to read.

When a process attempts to write to an opened file, it should grab the mutex lock for the other device, since that's the one being written to, then ensure it can write to the buffer, and if the buffer is full, it should put itself to sleep in the writing queue of that device. When it can write, it should take the content of a user space buffer and write it to the buffer of the device, again following the same rules about wrap-around as the read function.

Regarding ioctl, there should be methods to change the buffer size and change the max number of readers, but as a quality-of-life improvement, ability to get the current buffer size and current max number of readers and ability to revert to default values should also be implemented.

Implementation

The dm510 char driver struct was implemented according to the design plans, having two wait_queue_head_t instances, which is a simple linked list consisting of waiting processes, these are here called inq and outq, for readers and writers respectively, char pointers called buffer and end, which is defining the location of the buffer corresponding to the device, char pointers rp and wp, which are used to keep track of the state of the buffer, if it's full, if there's something to read and so forth, to integers, nreaders and nwriters, keeping track of the current number of readers and writers on the device, and finally we have the mutex and the char device structure as well, which are simply named mutex and cdev.

Load / unload

When the module is loaded, dm510_init_module is called, this function initializes the module by calling register_chrdev_region() to allocate a range of device numbers for the driver based on the major and minor numbers defined in the constants (major 255 and minor 0 and 1). If register_chrdev_region() fails, an error message is printed and the function returns the error code. If register_chrdev_region() succeeds, the function continues by allocating memory for the devices using kmalloc(), returning -ENOMEM if it fails, and initializing each device using dm510_setup_cdev() and their corresponding queues. Finally, the function returns 0 if initialization was successful.

The function dm510_setup_cdev() works by calling cdev_init() to initialize the cdev structure with the file operations structure dm510_fops, which is a struct that is needed to define what functions correspond to the different system calls. Afterwards, cdev_add() is called to add the device to the system. If cdev_add() fails, an error message is printed with the error code, and the name of the device that failed.

When the module is unloaded dm510_cleanup_module() is called. It first checks if there are any devices to release. If there is, it loops through each device and calls cdev_del() to delete the character device. It then frees the buffer associated with each device using kfree(). Finally, it frees the memory allocated for dm510_devices using kfree() and unregisters the device number range using unregister_chrdev_region().

Open / close

When the system call open is called on a device file, the dm510_fops define that dm510_open is responsible for opening the device. The function first acquires mutex locks for both this character driver and the other character driver (target) and returns with the errorcode -ERESTARTSYS in case it couldn’t acquire the mutex, which makes the kernel retry the system call. It then it checks if there are too many readers or writers and unlocks the mutex and returns -EBUSY if there is. If not, it calls redefinedbuffer() (explained below) which returns an error code if it fails to allocate memory for a buffer. If redefinedbuffer() succeeds, it increments the number of readers or writers and unlocks both mutexes before returning the opened file.

redefinedbuffer() is a failsafe method to make sure the buffers are always defined when opening a file. This is needed because it was chosen to free the buffer when closing a file, and therefore it is needed to make sure there is a buffer in place on both devices. It works by allocating memory for the buffers on both devices if they don’t exist, returning -ENOMEM if allocation fails, followed by setting the rp and wp to the start of the buffer and end to the end of the buffer. Returns 0 on success.

Likewise, to close a device file, dm510_fops define that dm510_release is responsible for this system call. The function first locks the mutex. Then it checks if the file mode includes read access. If it does, it decrements the number of readers. Similarly, if the file mode includes write access, it decrements the number of writers. If there are no more readers or writers, it frees up memory allocated for buffer by calling kfree() function, this was chosen to conserve memory, and not have massive files saved. If character devices needed to keep the content through closes, a different implementation would just need this one line removed, and in that case, if the buffers were defined in the initialization of the modules, the redefinedbuffer() method was also likely not needed. Finally, it unlocks the mutex and returns 0 indicating success.

Read / write

To handle the system call read, dm510_read() is implemented. The function first locks the mutex, again giving an error if it is not able to. Then it checks if there is any data to read by comparing read pointer and write pointer of the buffer. If there is no data to read, it waits for data to be written by another process by going to sleep using wait_event_interruptible() given the queue to wait in, and a condition as arguments. It will then sleep in this queue until the condition is true, it will check the condition each time it is woken up by another process. If there is data to read, it copies the data from the buffer to the user space buffer using copy_to_user() function. It then updates rp and unlocks the mutex. Finally, it wakes up any sleeping writers and returns the number of bytes read.

Like the previously described function, the dm510_write() is responsible for handling the corresponding system call. It takes in some data in a user space buffer and acquires the mutex lock for the other device, so no other process can write to it. It then calls dm510_getwritespace() to ensure there is room to write (explained below), if this doesn't succeed, it returns with the error message returned from this method. If this is successful spacefree() is then used to check how much space is free in the buffer (explained below) and it then either writes the complete message, or as much as the message as there is space for in the buffer. The function will then unlock the mutex, wake up any sleeping readers and return with the number of bytes written, and in case this is less than what was attempted to be written, the system call will try again to write the rest, and end up sleeping in a queue.

To ensure there is space to write in the buffer, dm510_getwritespace() is called. This function checks if the buffer is full with spacefree(), which compares the read pointer and write pointer, and returns the amount of free bytes in the buffer, if there is no free space, the mutex is unlocked, and the process is put to sleep in the writing queue, outq. As soon as there's space in the queue, the process is woken up, and continues on.

ioctl

Finally, IOCTL has been implemented in the dm510_ioctl function. This function checks if the command passed is valid before going into a switch statement that has different functionality depending on the command passed. The different functionality offered is:

- DM510_IOCRESET – Resets buffer size and max readers to default
- DM510_IOCSBUFFER – Sets the buffer to the value given
- DM510_IOCSREADERS – Sets the max number of readers to the value given
- DM510_IOCTBUFFERSIZE – Returns the current buffer size
- DM510_IOCTMAXREADERS – Returns the current number of max readers

Testing

For testing both tests using cat/echo and test using the write and read system calls in a script was used. There are tests for how the driver handles too many readers, too many writers, test to show how a process is sleeping / waiting for something to be written or read from the buffer, also showing what happens if a process writes too much to a device. There's also test of the driver working properly when reading and writing to a device, and there's a test handling the different IOCTL commands. Each test is accompanied by a timestamp to the test in the included video.

Timestamp	Test	Result
00:44	Too many writers	After writing too much to a buffer to ensure it hangs waiting to write, another thread tries to write as well, but is blocked. Success!
01:15	Too many readers	Two readers are opened, waiting for something to read, a third then tries to open, but is blocked. Success!
01:38	Waiting to read	An already open reader is waiting for input, as soon as a writer then writes to the device, it is output at the reader. Success!
02:05	Waiting to write	A process writes repeatedly to a device until the buffer is full, at which point it is waiting for space to write. As soon as a reading process reads from the buffer, the write is completed. Success!
02:34	Regular behavior (echo/cat)	A process writes to one device, and another reads from the other, and this is tried on both devices, and the output is as expected. Success!
02:54	Regular behavior (scripted)	The script test.c is ran which opens device 0, writes some data to it, closes it, opens device 1 and reads that data back from it, closes it, then opens device 1, writes some data to it, closes it, opens device 0 and reads back that data. All outputs are as expected. Success!
03:10	Change ioctl settings (scripted)	The script test2.c is ran, which opens device 0, reads the current buffer size, changes it to 40 (globally) and prints the new value, then reads the current max readers, changes it to 4, and then prints the new value before closing the file again. Seems like a success to me!
03:22	Confirm new buffer size	A long string is echoed to device 1 to confirm that it can indeed store all 36 characters, which it indeed can, made obvious by the fact that the writing process does not hang, waiting to complete the write, and the fact that the

		reading process writes out all 36 characters. Success!
03:55	Confirm new maxreaders	3 different reading processes are opened at the same time to confirm that max readers have indeed changed. Success!

Concurrency

Based on the tests of the program, it doesn't seem like there is any issues with multiple processes accessing the devices at the same time. The mutex locks put in place seems to be doing their job in protecting the critical sections of the code, and the variables that set a limit on the number of concurrent writers and readers seems to work just fine. If these didn't work there might have been race conditions, where trying to update the number of writers and readers from different processes might not have registered correctly and thus, we could have had more writers and readers than allowed. This might not be a huge issue, since the mutex locks should still have protected the devices, but it would go against the assignment. If the mutex locks didn't work as they should, we could have more problems, with processes overwriting each other's messages, meaning messages would get lost.

Conclusion

In the end, a working kernel module was implemented. This exposed two character devices to user space, meaning that user processes could access these two devices, and they then transferred data between them in kernel-space via the device driver.

The driver worked as intended, solving the producer/consumer problem, with implemented queues and mutex which made this driver thread safe and robust against race conditions, even when multiple processes accessed the devices at the same time.

Device control via IOCTL was also successfully implemented, giving the opportunity to both see and change the current buffer size and max readers, and to reset the values to default.

This was a fun project to work with and we learned a lot as a group.