Device Drivers

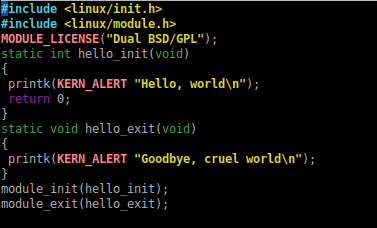
Vladislav Serafimov 509761

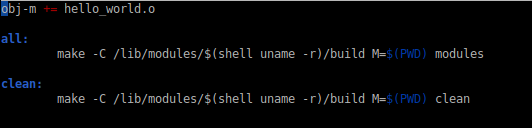
1. Introduction

I decided to read the book instead of following the tutorial, as it looked more organized. Chapter 3 and 6 are the most important for char drivers, so my focus was on them.

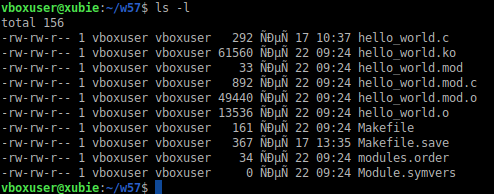
1. Hello world!

The first step to learning anything programming-related is writing a “Hello world!” program.

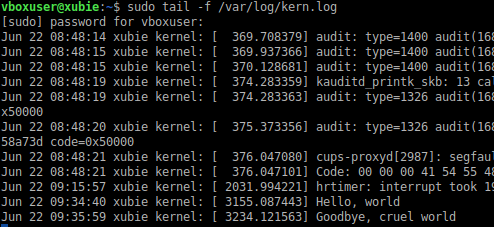




After compiling the code with that Makefile I got the kernel module *hello\_world.c*.

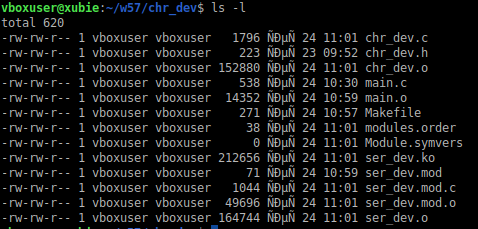


Inserting then removing the module returns the following



1. Base driver

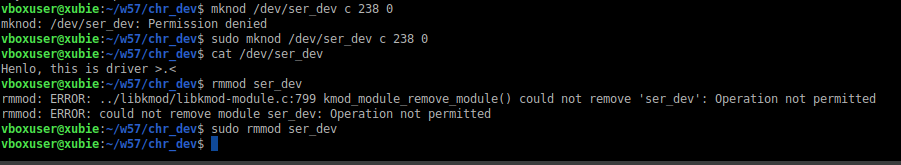
The base driver code is pretty much the same as the one in the tutorial, with some minor changes. After compiling it and naming the output *ser\_dev* I got the following as a result

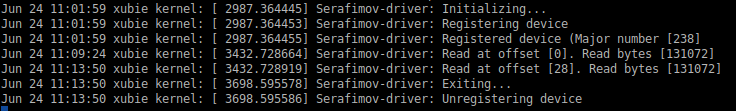






After that, I could use *mknod* to create a special character stream file and read the data.





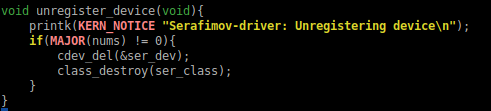
1. Extras
   1. Switching to *cdev*

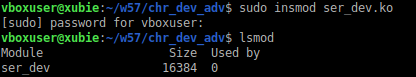
After making sure everything works, the next step is to optimize the basic functionalities. Here, we need to use the method which employs the *cdev* struct and its corresponding functions.

While we are at it, it would be a waste of time to keep using *mknode* when we have the option not to. This can be done by creating a device from the driver directly. Because we only need one I only create a device during initialization of the module and then release it during removal.

The only functions that needed changing were the *register\_device* and *unregister\_device* functions.

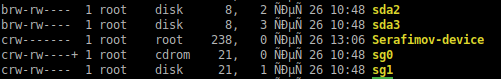




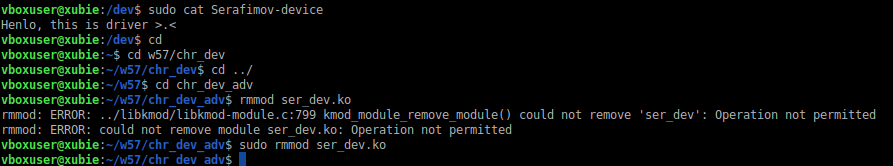


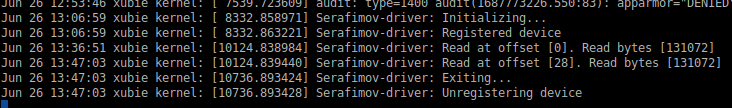


In */dev*



The other functionality has not changed so it behaves the same way after being inserted into the kernel.



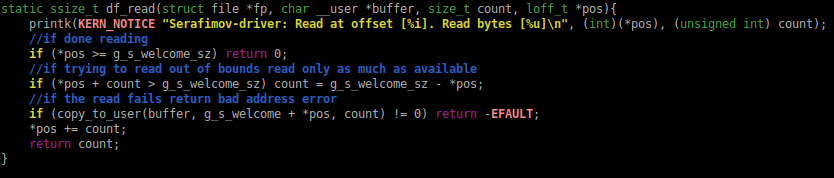


* 1. Implementing *write*

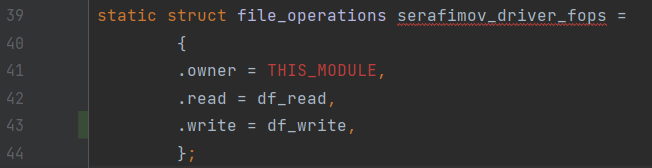
To implement *write*, there need to be some changes to *read*,too. The message string will have to be replaced with a non-constant buffer of arbitrary size.



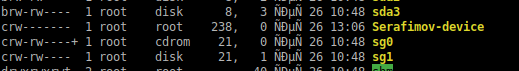
The *write* implementation is analogous to *read*

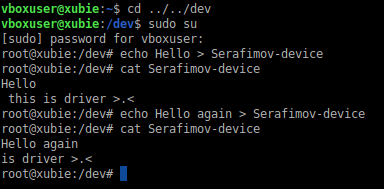


The function also needs to be added to the *fops*

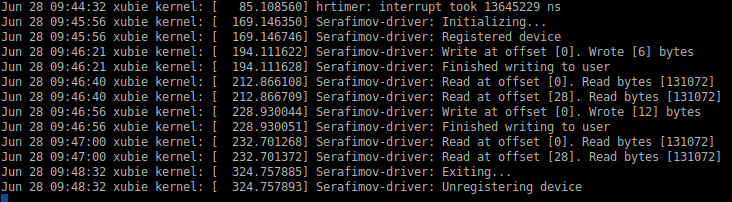


In */dev* the file subsystem tells us that our device can be written to and read from (which is correct)



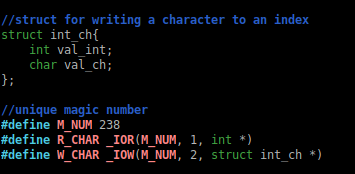


Checking the log we can see what actions happened



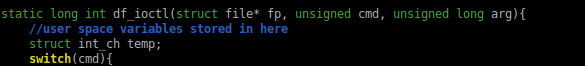
* 1. Adding *ioctl* support

*Ioctl* is used for custom commands. Lets say, I only need to read or write one character to the device file. Then I can implement a custom *ioctl* function which has a switch statement that, depending on the switch flag, executes something different. In my case that would be either reading or writing.

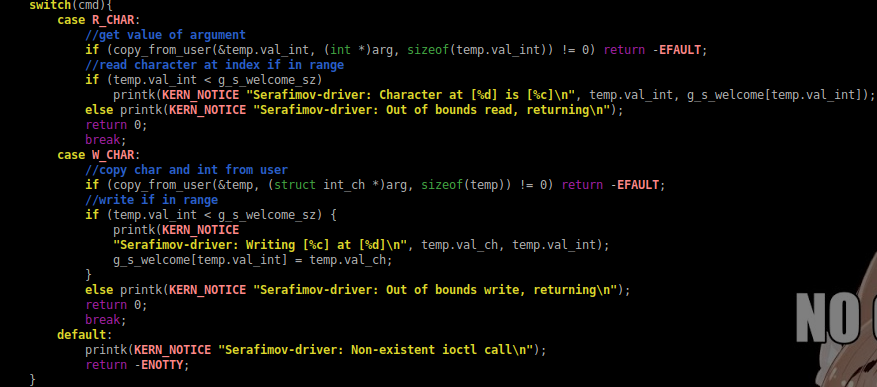


The struct is used because *ioctl* can pass only one argument. The *R\_CHAR* and *W\_CHAR* macros are the identifiers needed to select the right function and they use *\_IOR* and *\_IOW*, which are predefined and return a number.

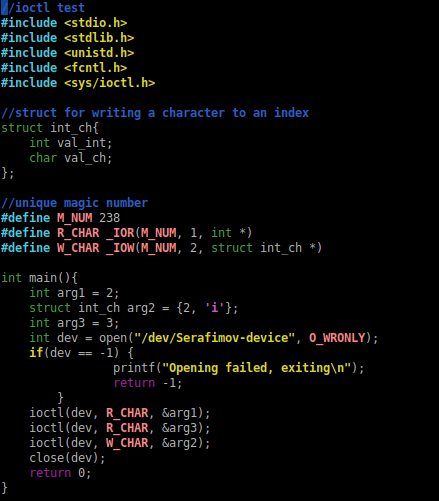
In the function definition *cmd* is the macro which will be passed and *arg* is the argument we will do some trickery on to get the value we need. *Temp* is needed to translate user pointers to kernel pointers.



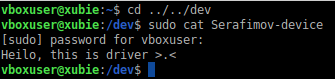
The functions just do that translation and then execute whatever functionality we need. If the user passes an unimplemented identifier, a not implemented error will pop up



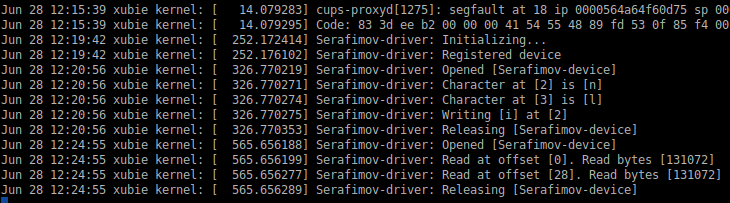
Testing is done with this program



After compiling and running the code we get the following.

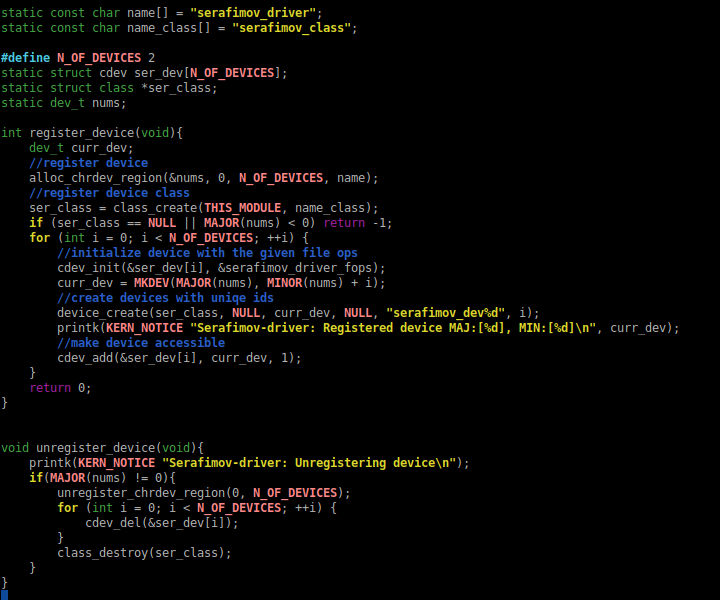


And the kernel log shows us what our user-space program did



* 1. Creating multiple devices

The current method of creating a device can easily be refactored to support multiple device creation (which was the purpose of adding a device class). The only functions that needed changing were the *register\_device* and *unregister\_device* functions. Most of the functions used were already meant for multiple devices so the change was not all that big. The code can use up to the maximum number of minor numbers for *N\_OF\_DEVICES*.



The testing code had to be slightly modified to use both files.



And because the buffer is shared the result is that operating on either device results in changing that one buffer.

