PA2 Solutions and Notes

Defined Definition of an LKM: “In computing, a loadable kernel module (or LKM) is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system. LKMs are typically used to add support for new hardware (as device drivers) and/or filesystems, or for adding system calls.”

Note* A key difference between adding a system call via an LKM is that you do not need to **recompile** your kernel. This is the idea of adding functionality on-the-fly.

Regarding major and minor numbers (most answers were incorrect):

Each device driver is given a major number. These major numbers are unique ids used to differentiate device drivers from each other within the Kernel. A single driver can run many different devices. So naturally minor numbers (associated with a single major number (driver)) would also be unique so as to differentiate between the different devices that driver is responsible for communicating with. Minor numbers may be used to represent 3 different partitions on a single hard drive, OR, they might represent 3 different hard drives. The idea is that minor numbers distinguish between devices (This is not solely related to partitions!).

Recall that everything in UNIX is modeled as a file. A proper definition: “A device file is an interface for a device driver that appears in a file system as if it were an ordinary file”. It is an interface for a device driver because it is an interface for communication with an actual physical device. Each device file will have two important numbers associated with it, namely, a major number and a minor number. The major number associated with the device file will be used to identify with driver will be responsible for communicating with the physical device, while the minor number will be used to identify and distinguish which particular physical device the driver will be sending data to/from.

Can multiple device files have the same Major number?

Yes, multiple device files can most certainly have the same Major number. As this would be representative of the fact that many device files utilize the same driver. In other words, a single driver might be responsible for communicating with

multiple physical devices on the same machine. (I.e. a single driver might be used to talk to all 3 of your Seagate Hard drives (and possibly each of the partitions within said hard drives).

Can multiple device files have the same Minor number?

Multiple device files may NOT have the same minor number (assuming they also share a major number), as this is used to distinguish physical devices from one another despite them using the same driver. The major-minor combo MUST be unique, and if majors can be shared, the minor numbers cannot be the same.

Extra Notes*

The file operations struct written inside of the driver module is used to tell the driver which operations a physical device (device file) will support. So if you are programming a device driver for a speaker, this will have different operations inside of the file operations struct than the file operations struct of a driver designed to work with a hard drive. Each of the operations supported (listed inside the struct) are then provided a function pointer which points to code on how to perform said operation on that physical device.

A user will request a read operation on a device file (which is representative of communicating with a physical device), the driver will see if that operation is supported inside of its file operation struct, if it is the driver will than proceed to execute the function that that operation holds a pointer to.

The edge cases (in hindsight, I could have worded this problem much better) we were looking for were considering the buffer sizes between the user buffer and device buffer.

The main edge case is the case that the user buffer size (length) < contents of the device buffer (NOT BUFFER_SIZE, rather strlen(device_buffer)). In which case to handle this, your read functions should not always be returning 0.

One piece of information regarding this. If a user requested a read operation on a device. The kernel/driver will call your read function again and again until it returns 0. Thus even if your user buffer was of size 1, ALL of the contents of the device file would still be printed. Thus you would read an amount of bytes = length (size of your user buffer), increment offset by the amount of bytes read. And keep reading and incrementing over and over again until the condition arises that (*offset == strlen(device_buffer), in which case you have read all the information inside of the device and would at this point return 0. There were other ways of achieving this using BUFFER_SIZE and noting the amount of bytes that failed to copy when using copy_to_user, and these solutions were also valid (although less intuitive and I am unsure if most of the students who used this solution actually understood why it worked). If you would like to see a solution, please do come to office hours.