ECGR 6181/8181 - Lab 4

Objective: Writing Linux device drivers

Outcomes:
After this lab, you will be able to

- Write a simple character driver for an “in-memory device” for ARM Linux
- Write a polling based UART driver for ARM Linux

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver. The kernel must have embedded in it a device driver for every peripheral present on a system.

Loadable modules
One of the features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running. Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time that we want new functionality.

Classes of devices and modules
The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module.

Character devices
A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially.

Block devices
Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. A block device can only handle I/O

operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

Network interfaces
Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets. Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Linux/Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission (socket programming).

In this Lab we deal with character devices

Majors and minors
Devices have unique identifiers associated with them. The identifier consists of two parts: major and minor. The first part identifies the device type (IDE disk, SCSI disk, serial port, etc.) and the second one identifies the device (first disk, second serial port, etc.). Most times, the major identifies the driver, while the minor identifies each physical device served by the driver. In general, a driver will have a major associate and will be responsible for all minors associated with that major.

Check these out on your system -
$ ls -la /dev/sd? /dev/ttyS?

The special character files are identified by the c character in the first column of the command output, and the block type by the character b. In columns 5 and 6 of the result you can see the major, and the minor for each device.

Character driver for a memory emulation of a character device

Included in the Google folder for Lab4, is the driver code (mem.c) that is loaded as a kernel module which runs in kernel space. Note that unlike user space programming, bugs in the kernel are major security risks, and can crash the machine.

The key parts of the driver code are -

- MODULE_LICENSE() macro which allows loadable kernel modules to declare their license to the world. Its purpose is to let the kernel developers know when a non-free module has been inserted into a given kernel.

Additionally, MODULE_AUTHOR and MODULE_DESCRIPTION macros can be added.

- module_init() and module exit() functions to register and unregister the driver with the kernel.

Our module_init() calls the memory_init() function, that uses register_chrdev() kernel function to register the driver with the kernel. A buffer is allocated in the kernel space using the kernel version of malloc() - kmalloc(), which is then zeroed out with memset(). printk() is a kernel version of printf(). Messages appear in the kernel log and are accessed via the dmesg command.
module_exit() calls memory_exit() which unregisters the device via unregister_chrdev(), and frees the buffer using kfree().

- Mapping of driver methods to file system calls.

To enable userspace to access the device using standard file system calls, each field in the file_operations structure must point to the function in the driver that implements the corresponding operation, or be left NULL for unsupported operations.
In our module we have the following mapping -

```
struct file_operations memory_fops = {
  read:  memory_read,
  write:  memory_write,
  open:  memory_open,
  release:  memory_release
};
```

memory_read() uses the copy_to_user() kernel function to transfer data from the kernel space buffer, to the userspace buffer. It updates the file offset to 1.

memory_write() uses the copy_from_user() kernel function to transfer data from userspace buffer to kernel space buffer.

Since we are emulating a character device in memory, memory_open() and memory_release() have no implementations.

Testing the memory character device

Included in the Google folder for Lab4, is the userspace test (mem_test.c) for the memory character device. After opening the character device /dev/memory" in binary write mode, we use standard C file operations to write and read from the device.

A Makefile is supplied to automate code compilation and building.

Create a Lab4 directory and download the files from the Google Lab4 folder. Open the Makefile, and adjust the kernel directory to that of the kernel path you downloaded in Lab3. From the Lab4 directory,

$ make

This generates the kernel module mem.ko and the test executable mem_test. Copy these to the rootfs directory created in Lab3

$ cd ../rootfs
$ cp ../Lab4/mem.ko .
$ cp ../Lab4/mem_test .
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../rootfs.cpio.gz
$ cd ..

Boot the kernel with QEMU
$ qemu-system-arm -M versatilepb -kernel linux-x.x.x/arch/arm/boot/zImage -dtb linux-x.x.x/arch/arm/boot/dts/versatile-pb.dtb -initrd rootfs.cpio.gz -serial stdio -append "root=/dev/mem serial=ttyAMA0"

To access the device, a file (which will be used to access the device driver) must be created, by typing the following command at the QEMU Linux prompt.

mknod /dev/memory c 60 0

In the above, c means that a char device is to be created, 60 is the major number and 0 is the minor number.

Now load the module,
insmod mem.ko

and unprotect the device with

chmod 666 /dev/memory

Execute test to check all the file operations (reading and writing of a single byte) is done from userspace (see mem test.c)
./mem_test

To do -
Write a Linux device driver for the UART terminals on the emulated QEMU VersatilePB board. This is a simplified character driver which reads and writes to the UART data register by polling; similar to the bare metal UART driver of Lab 2.

Test your driver by performing file I/O.
Hints:
1. On ARM all I/O access is memory mapped.

2. Use the kernel function ioremap in the uart initialization function to assign virtual addresses to the memory regions. Use the following physical address of UART0

#define UART0_ADDR 0x101F1000
uart_addr = ioremap(UART0_ADDR, 4096);

3. Use the kernel functions ioread8 and iowrite8 to read and write from the appropriate UART registers.

Take a look at the serial core driver in Linux just to get a feel like what a real-world Linux driver looks like -
https://github.com/torvalds/linux/blob/master/drivers/tty/serial/serial_core.c