

Architecture des Systèmes d'Exploitation Évolués

Université de Lille, Master 2 Info : A.A. 2023/2024

Giuseppe Lipari

September 9, 2023

Contents

1	Introduction	1
1.1	Compiling the kernel	2
1.1.1	WARNING	3
1.1.2	Debugging	3
1.1.3	Compiling	3
1.2	Debian Image	4
1.2.1	Prepare an image	4
1.3	Run the new kernel	4
1.4	Network problems	4
2	Linux kernel device programming	5
2.1	Part 1 : from "Hello world" to char devices	5
2.1.1	Compiling and executing hello-1.ko	5
2.1.2	Sycalls	5
2.1.3	Character devices drivers	6
2.1.4	From user space to kernel space	6
2.2	First TP	6
2.3	Part 2: Sysfs and ioctl	7
2.4	Second TP	7
2.5	Part 3: Blocking and sleeping	7
2.5.1	Task structure	8
2.5.2	Blocking Processes and threads	8
2.5.3	Mutexes	8
2.5.4	Spinlocks and atomic operations	8
2.6	Third TP: Blocking processes	9
2.7	Part 4: Interrupts	9
2.8	Fourth TP (optional): Interrupt handling	9

1 Introduction

The objective of this short course is to understand the internals of the Linux Kernel. More specifically we will see:

- How to set up a development environment for programming Linux kernel code on a PC (for embedded programming the tools to be used are slightly different);
- What is a kernel module, and how to write a simple one;
- How does scheduling works in Linux;
- How to debug and trace kernel code.

The course is not exhaustive; we will just touch the surface of some topic. However, it is a good starting point for people that would like to pursue the topic of kernel programming. It is also useful for the students interested in other topics because it gives an overview of the internal workings of the kernel, and of the many difficulties that you can find in developing system code.

In this course, we will use the on-line book "The Linux Kernel Module Programming Guide" (<https://sysprog21.github.io/lkmpg/>)

First, some information on how to download, compile and launch the kernel in a virtual machine.

1.1 Compiling the kernel

Download the latest version of the Linux kernel from <http://www.kernel.org>. Untar:

```
tar -xf linux-6.5.tar.xz
```

Prepare a directory `build/kvm/` and type

```
cd linux-6.5
make O=../build/kvm menuconfig
```

(it's an upper case letter 'O', not a 0). In the configuration menu, make sure the following options are set:

- Make sure you are compiling e1000 driver which we are going to use. It is in:

```
Device Drivers -> Network device support
-> Ethernet driver support
-> Intel(R) PRO/1000 Gigabit Ethernet support
```

- For serial console access you need

```
Device Drivers -> Character devices -> Serial drivers
-> 8250/16550 and compatible serial support
-> Console on 8250/16550 and compatible serial port
```

Other options for PCI or DMA are not necessary

- Make sure Virtualization is marked in main menu and KVM support is enabled for your processor (Intel or AMD). You can disable any options regarding the host.
- Mark

```
Device Drivers -> Virtio drivers
-> PCI driver for virtio devices
```

Then go to

Device Drivers -> Block devices

and mark Virtio block driver.

Pay attention: you need to mark these options with an asterisk "*" and not with a "M" (module), otherwise they will be compiled as kernel modules and their use is going to be more complex. We can set other virtio drivers too, but they are not mandatory for our course.

Typically, most of these options are already set, but please check out, then save the configuration file.

1.1.1 WARNING

In your kernel configuration file you will find these lines:

```
CONFIG_SYSTEM_TRUSTED_KEYS="debian/canonical-certs.pem"
CONFIG_SYSTEM_REVOCATION_KEYS="debian/canonical-revoked-certs.pem"
```

Change it to this:

```
CONFIG_SYSTEM_TRUSTED_KEYS=""
CONFIG_SYSTEM_REVOCATION_KEYS=""
```

Please, check that libssl-dev is installed in your system.

1.1.2 Debugging

It is useful to set the following config variable :

```
CONFIG_MODULE_FORCE_UNLOAD=y
```

this will allow you to unload a module even if something went wrong and the kernel thinks it is unsafe to unload the module. It may save you some reboot.

If you want to use gdb for debugging the kernel, the procedure is a little more complex. I suggest you look at the following link:

<https://www.josehu.com/memo/2021/01/02/linux-kernel-build-debug.html>

In this course, it is not necessary to use gdb, so you can skip the above.

1.1.3 Compiling

Then, go inside the target directory and type

```
cd ../build/kvm
make -j8 bzImage
make -j8 modules
```

The last step is necessary to generate the `Module.symvers` file that contains the exported symbols by the kernel and by all the modules with their CRC. Without this file it is very difficult to compile your own module, so this is unfortunately a mandatory step.

Please be advised that compiling the kernel can take **up to 1 hour**, so relax and continue reading the rest of the documentation while the kernel compiles.

1.2 Debian Image

I prepared a Debian image that you can download here. Then, to test if it works, you can run the script `kvm.sh` to run the image into a QEMU-KVM virtual machine.

Optional: If you want to prepare your own image, please follow the instructions below.

1.2.1 Prepare an image

Download a ISO from Debian.

Create an image (http://wiki.colar.net/creating_a_qemu_image_and_installing_debian_in_it)

```
qemu-img create -f qcow debian.qcow 2G
qemu -cdrom debian.iso -hda debian.img -boot d
```

You can use the script `kvm-prepare.sh` to install your own debian image, or you can use the one I prepared for you.

Please notice that you may need to install `sudo` and other packages in your debian image.

1.3 Run the new kernel

Once the kernel has been compiled, run the script `kvm-mykernel.sh` that you will find in this repository to launch the debian image with your kernel (instead of the standard one). You need to adjust the first line to point the location of your compiled kernel.

Then you can connect to the virtual machine using

```
ssh -p 10022 root@localhost
ssh -p 10022 asee@localhost
```

For my image, in both cases the password is `asee`.

1.4 Network problems

Sometimes the dhcp client does not work properly (problems at start time, maybe). The problem seems to have disappeared in the latest versions. However, if you encounter it, here is a solution:

<https://stackoverflow.com/questions/53199827/my-newly-compiled-kernel-loses-networking-in-qemu>
To solve the problem, in the guest OS you can run

```
ip a
dhclient -v <interface>
```

where interface is the one that corresponds to the ethernet link in the output of "ip a". To do this permanently, just add the following lines to the file `/etc/network/interfaces`

```
auto <interface>
iface <interface> inet dhcp
```

2 Linux kernel device programming

In this course, we will use the on-line book "The Linux Kernel Module Programming Guide".

<https://sysprog21.github.io/lkmpg/>

In this course, it is not necessary to read the whole book. In the following I will highlight the mandatory parts and the optional parts. Also, we will use some of the provided examples. I recommend you clone the github repository with the book and all the examples.

It is often useful to explore the kernel code and see what the functions do, what is their prototype and their definition, read the comments, etc. Since the kernel is huge, it is difficult to explore it conveniently without a support. You may use the following website to search the kernel tree and explore the code:

<https://elixir.bootlin.com/linux/latest/source>

2.1 Part 1 : from "Hello world" to char devices

This part covers Chapter 1 (Introduction), Chapter 4 (Hello World), Chapter 5 (Preliminaries) and Chapter 6 (Character Device Drivers).

2.1.1 Compiling and executing hello-1.ko

Go inside `work/hello-1` and read section 4 until 4.6 of the book (no need to cut and paste, the code is already in this repo) while looking at the code.

Please notice that, in the book the authors assume that you are compiling the module for the host (that is the kernel where you work and compile). Therefore, after compiling, they ask you to directly load the kernel with `insmod hello-1.ko`.

In our case, we are compiling in the host, but the kernel and the module will be executing in the target (the virtual machine). Therefore, after compiling, you need to copy the module into the target with the following command:

```
$ scp -P 10022 hello-1.ko asee@localhost:
```

and then you need to log in into the target to load the module:

```
$ ssh -p 10022 asee@localhost
asee@debian$ sudo su
root@debian:/home/asee# insmod hello-1.ko
```

Once you load the module you can watch the last 5 messages emitted by all modules with `printk` by running the command

```
root@debian:/home/asee# journalctl | tail -n 5
```

2.1.2 Sycalls

Read section 5 of the book to understand what is going on. In particular, run the example in section 5.2 to see the trace of syscalls.

2.1.3 Character devices drivers

Read Section 6 of the book until section 6.5.

Some additional comment: we may notice that the `file_operations` structure is a sort of "interface" for our character device. By using an **object-oriented analogy**, we can interpret the code as follows:

- The character device we are coding can be seen as an object of a class that "derives" from a generic class "DeviceDriver";
- The parent class "DeviceDriver" implements a set of virtual functions for opening the device driver (`open`), for reading from or writing to it (`read` and `write`), for moving the head (`llseek`), etc. Most of these functions are not implemented, that is they are abstract functions.
- Our device driver (the derived class) has to overload some of these functions to implement the desired behaviour. In other words it has to say what does it mean to read or to write to the device. To do this, it implements the corresponding functions and store their address into a `file_operations` structure. Then it registers the structure within the kernel to the corresponding device file. In our analogy, this is equivalent to overload the virtual functions.

Of course, since we are coding in C, we cannot use the typical constructs of an object-oriented language like C++, so the kernel developers use a structure of pointers to functions instead. You may notice the correspondence with the virtual function table (VTABLE) that is used in C++.

Why using such an interface? In Linux, a device driver is exposed to the user as a file in directory `/dev/`. Therefore, the `file_operations` structure lists all operations that may be performed on a file, and gives the programmer of a module the possibility to overload such functions to perform operations on the device (rather than on a classical file on a disk).

2.1.4 From user space to kernel space

The example in section 6.5 deals with a read-only character device. If we want to write to the device, we have to implement a `device_write()` function.

This is important: when in user space, we see memory differently than in kernel space. Remember the course ASA: virtual memory involves using translation tables that map virtual addresses to physical addresses: kernel and the user map the same physical address to different virtual addresses. Therefore, every time the user process passes an address to the kernel, it is necessary to do a "translation" of that address to be able to transfer the data.

To import data from user space, we need to use the function `copy_from_user` that takes an address in user space and copies the content into an address in kernel space. The reverse function is `copy_to_user`. If we have to just copy one single byte, we can use `get_user` and `put_user`.

2.2 First TP

By using the structure of the example in section 6.5, write a character device `/dev/asee_mod` that :

1. Stores the characters that the user writes into the device into a circular buffer of 16 characters
2. Reads the characters in the buffer in the same order they have been written.
3. Keeps the buffer alive between open and close operations.

In particular, we want to observe the following behaviour:

```
echo "Hello world" > /dev/asee_mod
```

will store "Hello world" into the buffer.

```
echo "Ciao" > /dev/asee_mod
cat /dev/asee_mod
```

will print the "Hello worldCiao" on the screen. A following

```
cat /dev/asee_mod
```

will print nothing (the data has been consumed).

If more than 16 characters are written in the buffer, then the first characters are overwritten (the buffer is circular). For example:

```
echo "abcdefghijklmnopq" > /dev/asee_mod
cat /dev/asee_mod
```

will print on screen "bcdefghijklmnopq" (the first 'a' has been overwritten). So, only the last 16 characters will be shown.

2.3 Part 2: Sysfs and ioctl

To interact with your module, you may need to change its configuration. We are going to use a different interface for it, the sysfs.

Read section 8 about the sysfs to know how to program this interface. Read, understand, compile and execute the example.

2.4 Second TP

The idea is to slowly transform our `asee_mod` device into a many-to-many communication channel between different processes. One process can write (produce) data to the channel, and other processes can later read (consume) the data. This is similar to the `pipe()` system call, however our channel is global and accessible to every process.

1. Starting from your code for TP1, add a variable `asee_buf_size` that contains the current buffer size (by default 16) and a variable `asee_buf_count` that contains the number of characters currently contained in the buffer. For debugging purposes, you may decide to add additional variables. These variables will all be contained in `sys/kernel/asee_mod/`.
2. The size of the buffer is now be a variable that can be changed by writing into `asee_buf_size`. Therefore, you should also modify the existing code to take this change into account. Pay attention that, when you decrease the buffer size, the new size could be less than the number of characters currently present inside the buffer: in this case, the operation is aborted, and the size is not modified. Also, you may log the error into the log file with `pr_err()` (see <https://www.kernel.org/doc/html/latest/core-api/printk-basics.html>).

Implement and test your module.

2.5 Part 3: Blocking and sleeping

Read section 11 of the book. Read, understand, compile and execute the example in 11.1.

Read section 12 of the book. Read, understand, compile and execute the examples.

2.5.1 Task structure

Inside the kernel, a thread (or a process) is called a task. All information about a task are contained in the task structure.

In this task structure there are many relevant things:

- the process id (pid)
- its exit state (in case the task is in zombie state)
- the amount of time it has executed (the utime/stime/gtime fields)
- the scheduling policy for this task
- its priority

and so many other things. One of the most important is the task **state**. A task can be in one of the following states:

- `TASK_RUNNING`, the task is executing
- `TASK_INTERRUPTIBLE`, the process sleeps waiting for an event or a signal
- `TASK_UNINTERRUPTIBLE`, the process waits for something, but it cannot be wake-up by a signal
- `TASK_STOPPED`, the task waits for a `SIG_CONTINUE`
- `TASK_TRACED`, the task has been suspended by a debugger
- `EXIT_ZOMBIE` and `EXIT_DEAD`, the task has finished executing, but the structure has not been deleted yet.

2.5.2 Blocking Processes and threads

When a thread needs to wait for an event, it can be blocked (put to sleep) by changing its status and inserting it in a waiting queue. See `modules/sleepmod.c` and the description here.

2.5.3 Mutexes

Kernel mutexes are very similar to userland mutexes. You can use them almost in the same way. See here for the definition.

In particular, `mutex_lock()` tries to lock the mutex, and if it fails, it blocks on a queue (i.e. its state becomes `TASK_UNINTERRUPTIBLE`, see here).

Mutex operations are easy to use but are internally complex, see for example the code of the most common case here. Therefore, use them when you have no stringent performance requirement.

2.5.4 Spinlocks and atomic operations

In case your critical section is very short (a few tenths of instructions), you may consider using a lower level mechanism called *spinlock*. A spinlock is a busy-wait on a condition, that is the task does not sleep. Very useful to avoid conflicts in multicore systems, must be avoided in single processor systems (use mutexes instead). Finally, for single operations on simple data (i.e. integers), consider using atomic operations.

2.6 Third TP: Blocking processes

We continue the work done in TP2, and we add the possibility to block the tasks under certain circumstances.

1. If the channel is full and a task wants to write additional data, it is **blocked** (sleeps) until there is at least one byte available for writing. This is different from before: in TP1 and TP2, when the buffer was full, a process would overwrite part of the buffer without blocking. Now, we require the writing process to block if there is not enough space.
2. In the first two TPs, if the channel was empty, the reader would return an empty string. Now, if the channel is empty and a task wants to read data from it, it is **blocked** (sleeps) until there is at least one byte available for reading.
3. Handling conflicts: Since several tasks can read/write at the same time executing in parallel on different cores, it is necessary to protect the data structures with mutexes or with spinlocks.
 - Use a single spinlock first for the whole device. Pay attention, you must release the spinlock before sleeping
 - Try to think about using a mutex. What does it change?

2.7 Part 4: Interrupts

Read Section 15 of the book.

We are developing in a virtual machine that simulates a PC, so we have no way for the moment to install a physical button on the GPIO. To simulate the occurrence of an interrupt, we can use the HW instruction `int`, as described here:

<https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-part-13-interrupts/>

2.8 Fourth TP (optional): Interrupt handling

Modify your previous code from TP3 so that, when a certain interrupt arrives, every blocked task is unblocked and returns with an error. To simulate the occurrence of an interrupt, you can implement a special `sysfs` variable that, when written, will raise the interrupt with `INT`.