Linux Kernel Programming

Redha Gouicem and Julien Sopena

Lecture 03:

Implementing Kernel Modules

A Quick Tour of the Kernel Configuration and Build System

Getting the Sources

From the official kernel website, **kernel.org**, download the *tarball* archive.

Or from the command line, for example:

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.5.7.tar.xz
```

You can (and **should**) also check out the integrity of the tarball with the pgp signature:

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.5.7.tar.sign
$ unxz linux-6.5.7.tar.xz    # the signature is done on the decompressed tarball
$ gpg --verify linux-6.5.7.tar.sign linux-6.5.7.tar
```

This will probably fail because you don't have the public keys of the maintainers that generated the tarball.

Get them from the kernel's key server (documentation):

```
$ gpg2 --locate-keys torvalds@kernel.org gregkh@kernel.org
```

You can also clone Linus Torvalds' git tree:

```
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/
```

Configuring the Kernel

The kernel configuration describes the features that will be enabled in the built binary, as well as change their behavior. It also describes if features should be built-in the binary or compiled as modules.

By default, the `Makefile`-based build system uses the file `.config` located at the root of the kernel sources.

You can generate initial configurations with the following commands (non-exhaustive list):

```
$ make allnoconfig     # minimal, everything that can be disabled is disabled
$ make defconfig       # default configuration for the local architecture
$ make localmodconfig  # configuration based on the current state of the machine (plugged devices, etc.) and builds them as modules
$ make localyesconfig  # same but everything is built-in
$ make oldconfig       # keeps the values of the current .config and asks for the new options
```

Or you can copy the configuration of your running kernel:

```
$ cp /boot/config-$(uname -r)* .config  # available on some distros
$ zcat /proc/config.gz > .config        # available if CONFIG_IKCONFIG_PROC is enabled
```

If you need to know more about your hardware to generate your config, check out these commands:

- `lshwd`, `lscpu`, `lspci`, `lsusb`, ...
- `dmidecode`
- `hdparm`
- `cat /proc/cpuinfo`, `cat /proc/meminfo`, ...
- `dmesg`

Building the Kernel

The kernel build system is based on `Makefile`s.

Just run `make` to compile it.

```
$ time make
```

```
real 80m15.486s
user 74m54.606s
sys 5m32.300s
```

Compilation can take a long time, so do it in **parallel**!

```
$ make -j $(nproc)
```

The compilation produces the following important files:

- `vmlinux`: the raw Linux kernel image. This ELF is used for debugging and profiling;
- `System.map`: symbol table of the kernel. Not necessary to run the kernel, used for debugging;
- `arch/<arch>/boot/bzImage`: compressed image of the kernel. This is the one that will be loaded and used.

```
$ du -sh vmlinux arch/x86/boot/bzImage
```

```
49M vmlinux
13M arch/x86/boot/bzImage
```

You can get some info on the image with the `file` command:

```
$ file arch/x86/boot/bzImage
```

```
arch/x86/boot/bzImage: Linux kernel x86 boot executable bzImage, version 6.5.7-lkp (redha@wano) \
#2 SMP PREEMPT_DYNAMIC Thu Oct 19 16:05:37 CEST 2023, RO-rootFS, swap_dev 0XC, Normal VGA
```

Installing a Kernel

Two main steps:

1. Install the kernel image, the symbol map and the initrd

```
$ make install
```

This will copy the image and symbol map in `/boot`, and generate the *initramfs*.

2. Install the modules

```
$ make modules_install
```

This will copy the modules (`.ko` files) into `/lib/modules/<version>`.

About the Symbol Map

The symbol map (`System.map`) provides the list of the symbols available in this kernel, their address and type.

```
$ head System.map
0000000000000000 D __per_cpu_start
0000000000000000 D fixed_percpu_data
0000000000001000 D cpu_debug_store
0000000000002000 D irq_stack_backing_store
0000000000006000 D cpu_tss_rw
000000000000b000 D gdt_page
000000000000c000 d exception_stacks
0000000000014000 d entry_stack_storage
0000000000015000 D espfix_waddr
0000000000015008 D espfix_stack
```

Check the manpage of the `nm` program for an explanation of the types.

As a rule of thumb (mostly true), lowercase means local scope while uppercase means global scope (*i.e., exported symbol*).

Linux Init Process

Linux Kernel Modules

Development Infrastructure

Multiple development methods:

- **Local setup**
  Use your usual development software, compile and run your new modules/kernel on your machine.
  Pros: easy, quick
  Cons: if a crash occurs, you can do nothing
- **Remote machine**
  If you have access to a separate testing machine, you can do your development on your machine and test remotely to avoid the crash issues. This machine is usually hooked through network/serial to the development machine to allow remote debugging and monitoring.
  Pros: good development setup, robust to crashes
  Cons: not always possible to have a second machine
- **Virtual machine**
  Develop on your local setup and deploy on a virtual machine. This replaces the previous method well, while being faster to use, and doesn't require a second machine.
  Pros: good development setup, robust to crashes, single machine
  Cons: doesn't always perfectly capture real hardware, might be slow depending on the host and guest machines

> **Note**
>
> In this course, we will use the last method with QEMU as a hypervisor.

Kernel Modules Interface

A **module** is a library dynamically loaded into the kernel. It triggers a call to a registered function when loaded and when unloaded.

The kernel provides two macros to register these functions: `module_init()` and `module_exit()`.

```c
static int my_init(void)
{
	/* ... */
	return 0;
}
module_init(my_init);
```

```c
static void my_exit(void)
{
	/* ... */
}
module_exit(my_exit);
```

For these to work, you will need some header files included:

```c
// contains the module API
#include <linux/module.h>

/// contains the init and exit macros
#include <linux/init.h>

/// if needed: base types, functions, macros...
#include <linux/kernel.h>
```

Module Information

You should also add some information about your module with some pre-defined macros, usually at the beginning of the file:

```
1 MODULE_DESCRIPTION("Hello world module");
2 MODULE_AUTHOR("Redha Gouicem, RWTH");
3 MODULE_LICENSE("GPL");
```

These can be checked on any module:

```
$ modinfo hello.ko
```

```
filename: hello.ko
description: Hello World module
author: Redha Gouicem, RWTH
license: GPL
vermagic: 6.5.7-ARCH 686 gcc-13.2.1
depends:
```

> ⚠ **Warning**
>
> The license is not only informative. It is also used to check if you are allowed to use some symbols in the kernel.

Example: Hello World

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

MODULE_DESCRIPTION("Hello world module");
MODULE_AUTHOR("Redha Gouicem, RWTH");
MODULE_LICENSE("GPL");

static int __init hello_init(void)
{
	pr_info("Hello World!\n");

	return 0;
}
module_init(hello_init);

static void __exit hello_exit(void)
{
	pr_info("Goodbye World...\n");
}
module_exit(hello_exit);
```

Annotations

The `__init` and `__exit` annotations are used to help the compiler optimize the memory usage.

When some module is statically built-in the kernel binary, functions tagged with these annotations are placed in specific segments:

- `.init.text` that is freed after the boot of the kernel
- `.exit.text` that is never loaded in memory

Building a Module

The running kernel is deployed with a generic `Makefile` located in `/lib/modules/$(uname -r)/build`.

You can use it from anywhere like this:

```
$ make -C /lib/modules/$(uname -r)/build M=$PWD
```

This will generate your module as a `.ko` file (*kernel object*).

You can also use a custom `Makefile` like this one as a wrapper:

```
1  ifneq ($(KERNELRELEASE),)
2
3    obj-m += hello.o
4
5  else
6
7    KERNELDIR_LKP ?= /lib/modules/$(shell uname -r)/build
8    PWD := $(shell pwd)
9
10 all:
11 ______make -C $(KERNELDIR_LKP) M=$(PWD) modules
12
13 clean:
14 ______make -C $(KERNELDIR_LKP) M=$(PWD) clean
15
16 endif
```

Loading/Unloading a Module

Loading a module can be done with `insmod`:

```
$ insmod hello.ko
$ dmesg
```

```
[177814.017370] Hello World!
```

Unloading a module can be done with `rmmod`:

```
$ rmmod hello
$ dmesg
```

```
[177919.956567] Goodbye World...
```

Module Parameters

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

static char *month = "January";
module_param(month, charp, 0660);

static int day = 1;
module_param(day, int, 0000);

static int __init hello_init(void)
{
      pr_info("Hello ! We are on %d %s\n", day, month);
      return 0;
}
module_init(hello_init);

static void __exit hello_exit(void)
{
      pr_info("Goodbye, cruel world\n");
}
module_exit(hello_exit);
```

With default values:

```
$ insmod hello.ko
$ dmesg
```

```
[180525.067016] Hello ! We are on 1 January
```

With parameters:

```
$ insmod hello.ko month=December day=31
$ dmesg
```

```
[181086.216097] Hello ! We are on 31 December
```

Kernel Dynamic Linker

Like shared libraries, modules are dynamically loaded: **they only have access to symbols explicitly exported to them!**

By default, they have access to absolutely **no** variable or function from the kernel, even if they are not `static`!

Two macros allow to explicitly export symbols to modules:

- `EXPORT_SYMBOL(s)` makes the symbol `s` visible to all loaded modules
- `EXPORT_SYMBOL_GPL(s)` makes the symbol `s` visible to all modules with a license compatible with GPL (according to their `MODULE_LICENSE`)

Example: using the `pm_power_off()` function exported in `arch/x86/kernel/reboot.c` and available on my system:

```
$ grep pm_power_off /lib/modules/$(uname -r)/build/System.map
```

```
ffffffff810ed2f0 t legacy_pm_power_off
ffffffff8274d7d8 r __ksymtab_pm_power_off
ffffffff838a47f8 B pm_power_off
```

```
 1 #include <linux/module.h>
 2 #include <linux/kernel.h>
 3
 4 MODULE_DESCRIPTION("Power off module");
 5 MODULE_LICENSE("GPL");
 6
 7 static int __init devil_init(void)
 8 {
 9       pr_info("The end is nigh...\n");
10       if (pm_power_off)
11             pm_power_off();
12
13       return 0;
14 }
15 module_init(devil_init);
```

Module Dependencies

If a module *X* uses at least one symbol from module *Y*, then *X* **depends on** *B*.

Dependencies are not explicitly defined: they are automatically inferred when compiled and available at `/lib/modules/<version>/modules.dep`.

This file is generated by `depmod`.

You can also check the dependencies of a module with `modinfo`.

Automated dependency solving

Obviously, modules must be inserted in the proper order: if *X* depends on *Y*, *Y* needs to be inserted before *X*.

If you are using `modprobe`, it will automatically insert dependencies first.

This is also true for unloading modules (in the reverse order).