Linux kernel and C programming

BLOCK 2: Linux kernel modules

Gergely Korcsák

Óbuda University
korcsak.gergely@nik.uni-obuda.hu

March 11, 2025

Presentation Overview

C Leftover shorts

goto

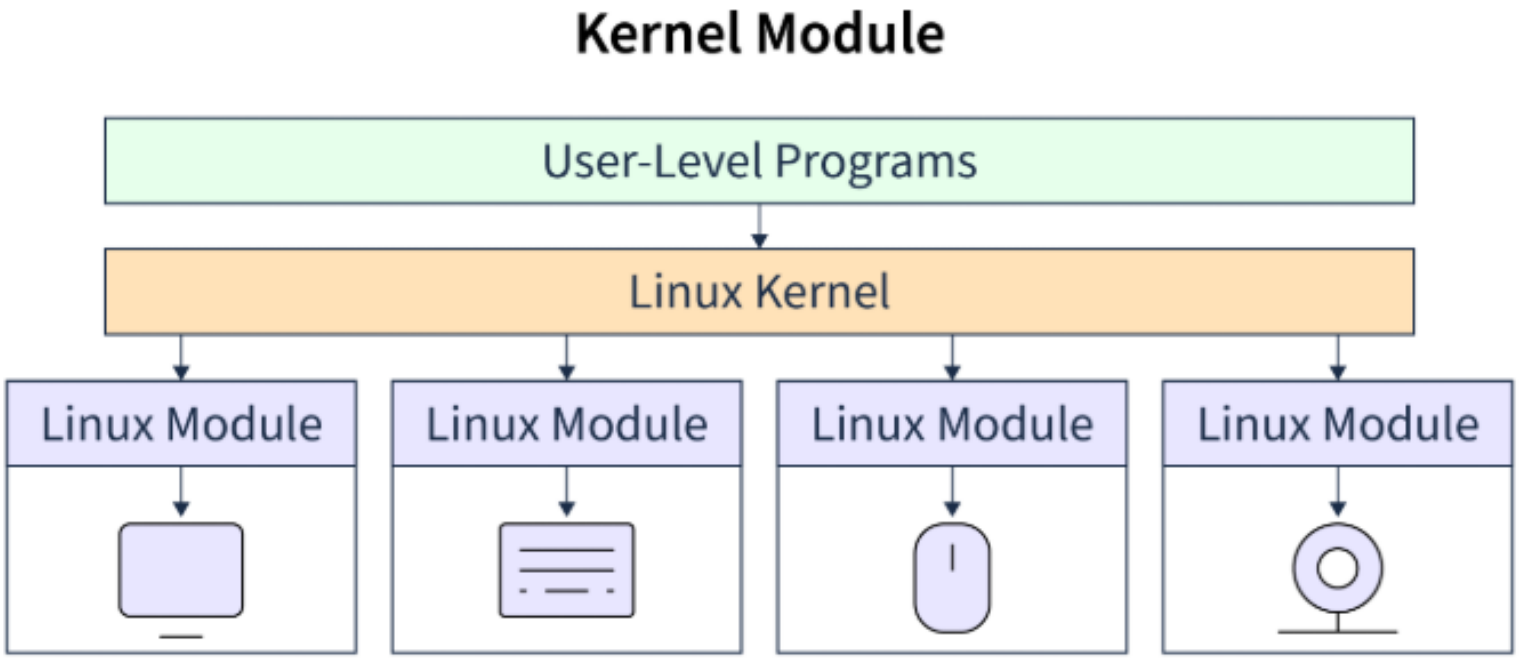
goto

With the `goto` keyword, we can jump to a specific line to continue execution from.

```
1  int main() {
2      printf("hello 1");
3
4      goto section2;
5
6      printf("hello 2"); // Will not execute
7
8  section2:
9      printf("hello 3");
10
11     return 0;
12 }
```

Linux kernel modules

Introduction



Headers

To build modules, first we will need to get the kernel headers.

```
$ sudo apt install build-esseitials kmod \
                   linux-headers-$(uname -r)
```

With which we can now access the kernel functions, with dynamic linking.

Hello World

We can create our first module:

```c
// hello_module.c
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

int init_module(void)
{
    pr_info("Hello world\n");

    return 0;
}

void cleanup_module(void)
{
    pr_info("Goodbye world\n");
}

MODULE_LICENSE("GPL");
```

Building

We can build it with the following `Makefile`

```
# Makefile
# We need to add the C files as object files
obj-m += hello_module.o

BUILDDIR  := $(shell pwd)/build
SRCDIR    := $(shell pwd)
KERNELDIR := /lib/modules/$(shell uname -r)/build

all:
    $(MAKE) -C $(KERNELDIR) M=$(BUILDDIR) src=$(SRCDIR)
    modules

clean:
    $(MAKE) -C $(KERNELDIR) M=$(BUILDDIR) src=$(SRCDIR)
    clean
```

Then build with `make all`.

Inclusion

modinfo

We can check the module information:

```
$ modinfo build/hello_module.ko
filename:       /.../build/hello_module.ko
version:        0.1
descrtiption:
author:         Bob
...
```

Inclusion

lsmod

Show the currently running modules

```
$ lsmod
ahci                23756  3
libahci             65734  1 ahci
sha1_ssse3          35664  1
...
```

Inclusion

insmod

Include a new module by path

```
1 $ sudo insmod build/hello_module.ko
```

Lets check it with lsmod

```
1 $ lsmod | grep hello
2 module_hello    12288  0
```

rmmod

And remove it with

```
1 $ sudo rmmod hello_module
```

Inclusion

dmesg

We can check the output of a module in dmesg

```
1 $ dmesg | tail -2
2 [ 2645.234345] Hello World!
3 [ 2646.234345] Goodbye World!
```

Init module with macros

We need to modify our code a bit. Lets include the `init.h` header as well, and specify the init functions by macros:

```
1  // ...
2  #include <linux/init.h>
3  // ...
4
5  // Modify init_module(void) to
6  static int __inti hello_init(void) {
7      // ...
8
9  // And cleanup_module(void) to
10 static void __exit hello_exit(void) {
11     // ...
12
13 // And add module_init(...) and module_exit(...)
14 module_init(hello_init);
15 module_exit(hello_exit);
```

Further module description

To build a module, we must define its license with `MODULE_LICENSE("GPL");` for example. But we can also specify the author and the description with:

```
1 MODULE_LICENSE("GPL");
2 MODULE_AUTHOR("LKMPG");
3 MODULE_DESCRIPTION("A sample driver");
```

Lets add our description, and check it with `modinfo`.

Proc files

Proc files were originally created to access information. They are located at the `/proc` folder. These files can be created when loading

a module, and its the modules job to remove them afterwards. Proc also has some useful systemfiles, like `meminfo` and `cpuinfo`.

Proc files

To create our own proc file, we need to include:

```
#include <linux/proc_fs.h>
#include <linux/uaccess.h>
```

Then fill create our functions for:

```
// coming from proc_fs.h
struct proc_ops {
    ssize_t (*proc_read)(struct file *, char __user *,
                         size_t, loff_t *);
    ssize_t (*proc_write)(struct file *,
                          const char __user *,
                          size_t, loff_t *);
    int (*proc_open)(struct inode *, struct file *);
    int (*proc_release)(struct inode *, struct file *);
    // ...
};
```

Proc files

Then crate our own struct

```
1 // create our own struct
2 static struct proc_ops my_proc_file {
3     .proc_read = my_proc_read,
4     .proc_write = my_proc_write,
5 };
```

Proc files - read

```
1  static ssize_t my_proc_read(
2              struct file* fp, char __user* buffer,
3              size_t buffer_length, loff_t* offset)
4  {
5      char s[] = "Hello World!\n";
6      if (*offset >= sizeof(s) ||
7          copy_to_user(buffer, s, sizeof(s)))
8      {
9          pr_info("copy_to_user failed\n");
10         return 0; // copy failed, written 0 bytes
11     }
12     else
13     {
14         pr_info("procfile read %s\n",
15                 fp->f_path.dentry->d_name.name);
16         *offset += sizeof(s);
17         return sizeof(s);
18     }
19 }
```

Proc files - write

```
1  static ssize_t my_proc_write(
2              struct file* fp, char __user* buffer,
3              size_t length, loff_t* offset)
4  {
5      char copy_buf[100];
6      size_t cp_size = length;
7      if (sp_size > sizeof(copy_buf))
8          cp_size = sizeof(copy_buf) - 1;
9
10     if (copy_from_user(copy_buf, buffer, cp_size))
11         return -EFAULT;
12
13     copy_buf[cp_size] = '\n';
14     *offset += cp_size;
15     pr_info("procfile write: %s\n", copy_buf);
16
17     return cp_size;
18 }
```

Proc files

Then we can create the file with

```
1 static struct proc_dir_entry* our_proc_file;
2
3 // in init(..)
4 our_proc_file = proc_create("hello", 0644, NULL,
5                             &my_proc_file);
6 if (NULL == our_proc_file) {
7     pr_alert("Error:Could not initialize /proc/hello\n");
8     return -ENOMEM;
9 }
```

And remove it with

```
1     // in exit(..)
2     proc_remove(our_proc_file);
```

Excercises

1. Lets create our first hello_module example
2. Create a proc file, to which we can write to, and read out its value. If we are out of buffer space, then return an error.

References

1. https://sysprog21.github.io/lkmpg/
2. https://elixir.bootlin.com/linux/v6.13.5/source/include/linux/