Kernel Object

It's like an object file, but for kernel.

Jules Aubert

2023

Get ready

```
apt install linux-headers-$(uname -r)
```

Might be something else on other distros.

Simple file - hello.c

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

static __init int hello_init(void)
{
    pr_info("hello: Hello World!\n");
    return 0;
}

static __exit void hello_exit(void)
{
    pr_info("hello: Goodbye.\n");
}

module_init(hello_init);
module_exit(hello_exit);
// MODULE_INFORMATION
```

Simple file - Module information

```
MODULE_LICENSE("Beerware"); // Mandatory
MODULE_DESCRIPTION("Hello module"); // Not mandatory
MODULE_AUTHOR("Jules Aubert"); // Not mandatory
```

About the pr_info function

printk

Simple file - Makefile

```makefile
KERNELDIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)
obj-m = hello.o

modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) $@

clean:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) $@
```

modinfo

```
$ modinfo ./hello.ko
```

Load the module into the kernel

```
$ sudo insmod ./hello.ko
```

List the modules

```
$ lsmod
```

You can group yours

```
$ lsmod | grep hello
```

Unload the module from the kernel

```
$ sudo rmmod hello
```

Kernel ring buffer

You can use dmesg(1) to get the logs.

```
$ sudo dmesg -L -T -W
```

The logs appear at runtime, when using the printk function.

- -L: put colors on the output
- -T: display time in human-readable format
- -W: only display new logs (not the one generated since the machine boot)

Multiple files

How to compile several source files into one kernel object?

Multiple files - The Makefile for several files

```
KERNELDIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)
obj-m = mult.o # IMPORTANT: It has to be named after the following variable!
mult-objs = kernel_test.o second.o

modules: $(OBJS)
	$(MAKE) -C $(KERNELDIR) M=$(PWD) $@ $^

clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) $@
```

Multiple files - The files

kernel_test.c second.c second.h

They’re a bit long, let me show the from my vim (or in the attached archive if you are reading this on your computer)

Arguments

How to put arguments when loading your object?

Arguments - The code 1/2

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

static int age = 42;
static char *login = "login_x";
static int grades[3] = {0};
static int grd_item = 0; // number of item in grades

module_param(age, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(age, "Your age");

module_param(login, charp, 0660);
MODULE_PARM_DESC(login, "Your login");

module_param_array(grades, int, &grd_item, 0660);
MODULE_PARM_DESC(grades, "Your grades");
```

Module param types

https://github.com/torvalds/linux/blob/master/include/linux/moduleparam.h#L120

```
find /dev/modules/ -type f -name 'moduleparam.h' 2>/dev/null
```

Arguments - The code 2/2

```c
static __init int hello_init(void)
{
    int i = 0;

    pr_info("args: Hello World!\n");
    pr_info("args: My login is %s and I am %d years old!\n", login, age);

    pr_info("args: args equals %d\n", grd_item);
    for (; i < grd_item; ++i)
        pr_info("args:\tgrades[%d] equals %d\n", i, grades[i]);

    return 0;
}

static __exit void hello_exit(void)
{
    pr_info("args: Goodbye %s.\n", login);
}
```

Arguments - The code 3/2

Never forget

```
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

Arguments - Loading without the args

If you don't put the args, or don't put at least one, it will take the default value in the code.

Beware with the array, it can't have more item than declared in the code. 3 here.

Arguments - Loading with args

Use all the args:

```
insmod ./args.ko login="aubert_o" age=29 grades=7,11,19
```

Use only some of them:

```
insmod ./args.ko grades=11,9 age=29
```

The order is not important.

Thread

How to add a thread into your kernel?

It is useful when you want to kernel to give you back the keyboard once the module in inserted if you are using a loop

Thread - The problem

Let's consider this snip of code

```c
static __init int story_init(void)
{
    pr_info("story: insmoded\n");
    while (1)
        pr_info("story: logging\n");

    return 0;
}

static __exit void story_exit(void)
{
    pr_info("story: rmmoded\n");
}
```

Exec - The problem

Once you **insmod** the module, you cannot get your keyboard back due to the fact that the **init** function never ends

It is a *Never Ending Story*

Thread - The solution

If you want to use an infinite loop, then you need a thread

Thread - The code

```c
#include <linux/module.h>
#include <linux/kthread.h>

static struct task_struct *thread = NULL;

static int threaded_func(void *data)
{
    char *kdata = data;
    while (kthread_should_stop() == 0)
        pr_info("thread: %s: %s\n", thread->comm, kdata);

    return 0;
}
```

Thread - The code

```c
static __init int thread_init(void)
{
    thread = kthread_run(threaded_func, "data", "thread_%d", 42);

    pr_info("thread: insmoded\n");

    if (IS_ERR(thread))
    {
        pr_err("thread: faield to create a kthread\n");
        return PTR_ERR(thread);
    }

    pr_info("thread: kthread started\n");

    return 0;
}
```

Thread - The code

```
static __exit void thread_exit(void)
{
    if (thread)
    {
        kthread_stop(thread);
        pr_info("thread: kthread stopped\n");
    }

    pr_info("thread: rmmoded\n");
}
```

Thread - About the threaded function

The kthread_run function takes a pointer to function with this exact signature:

```
int func_name(void *data);
```

This is why **threaded_func** uses this signature

Thread - About the kthread_run function

In the parameters of kthread_run, you can put data of any type, thanks to the pointer to void in the parameter of the threaded function

In the example I am using a pointer to char, but you can create your own struct and send the pointer to this struct from the **init** function to the threaded function

The last parameter is not really **42** but **thread_%d**, because the function is a **variadic** function (like printf(3)). So it will be formatted to thread_42

This parameter is used on the **comm** attribute of the struct task_struct

Thread - struct task

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

```
/lib/modules/$(uname -r)/build/include/linux/sched.h
```

Exec

How to exec code from the kernel?

We will exec **ls(1)** on a file given as a parameter

Exec - headers

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

Exec - Mandatory

```c
static int __init execls_init(void) {
    pr_info("execls: insmoded\n");
    return exec_ls();
}

static void __exit execls_exit(void) {
    pr_info("execls: rmmoded\n");
}

module_init(execls_init);
module_exit(execls_exit);

MODULE_LICENSE("GPL");
```

Exec - Global variables

```c
static char *output_file = "/tmp/execls_output";
static char *ls_file = "/tmp";
module_param(ls_file, charp, 0644);
MODULE_PARM_DESC(ls_file, "File to list");
```

Exec - exec_ls

```
static int exec_ls(void)
{
```

Exec - exec_ls

```c
struct subprocess_info *sub_info = NULL;
struct file *file = NULL;
char *cmd = NULL;
char *envp[] = { "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
int status = 0;
char *buf = NULL;
loff_t pos = 0;
int len = 0;

pr_info("execls: running ls on: %s\n", ls_file);
```

Exec - exec_ls

```c
cmd = kmalloc(4096, GFP_KERNEL);
if (!cmd)
{
    pr_err("execls: failed to allocate memory for cmd\n");
    return 1;
}
sprintf(cmd, "ls %s > %s", ls_file, output_file);
char *argv[] = { "/bin/sh", "-c", cmd, NULL };
```

Exec - exec_ls

```
sub_info = call_usermodehelper_setup(argv[0], argv, envp, GFP_KERNEL,\
                                     NULL, NULL, NULL);
if (sub_info == NULL)
{
    pr_err("execls: failed to setup usermodehelper\n");
    kfree(cmd);
    return 1;
}
```

Exec - exec_ls

```c
status = call_usermodehelper_exec(sub_info, UMH_WAIT_PROC);
pr_info("execls: finished with exit status: %d\n", status);
buf = kmalloc(4096, GFP_KERNEL);
if (!buf)
{
    pr_err("execls: failed to allocate memory for buffer\n");
    kfree(cmd);
    return 1;
}
```

Exec - exec_ls

```c
file = filp_open(output_file, O_RDONLY, 0);
if (IS_ERR(file))
{
    pr_err("execls: failed to open file: %ld\n", PTR_ERR(file));
    kfree(buf);
    kfree(cmd);
    return 1;
}
```

Exec - exec_ls

```
len = kernel_read(file, buf, 4096, &pos);
if (len < 0)
{
    pr_err("execls: failed to read file\n");
    filp_close(file, NULL);
    kfree(buf);
    kfree(cmd);
    return 1;
}
else if (len == 4096)
{
    pr_err("execls: buffer to read output file is too short\n");
    filp_close(file, NULL);
    kfree(buf);
    kfree(cmd);
    return 1;
}
```

Exec - exec_ls

```
pr_info("execls: output:\n%s\n", buf);
filp_close(file, NULL);
kfree(buf);
kfree(cmd);
return 0;
}
```

Exec - Test

```
$ sudo dmesg -T -W -L # Shell 1
...
$ sudo insmod execls.ko ls_file="/tmp" # Shell 2
```

Exec - TODO

Loop on the reading of the output file in case the file is larger than 4096 bytes

Handle stderr

Handle other shell executions

Network

To help you begin with the project, I will show you how to create socket, then you are on your own

Interesting page: https://docs.kernel.org/networking/kapi.html - Linux Networking and Linux Devices APIs

Network - Headers

```c
#include <linux/inet.h>
#include <linux/module.h>
#include <linux/net.h>
```

Network - Mandatory

```c
static int __init network_init(void)
{
    int ret = 0;

    pr_info("network: insmoded\n");

    ret = connect_to_server();
    if (ret == 0)
        send_message();

    return ret;
}
```

Network - Mandatory

```
static void __exit network_exit(void) {
    if (sock)
    {
        sock_release(sock);
        pr_info("network: socket closed\n");
    }

    pr_info("network: rmmoded\n");
}

module_init(network_init);
module_exit(network_exit);

MODULE_LICENSE("GPL");
```

Network - Global variables

```c
static char *ip = "127.0.0.1";
static int port = 4242;
module_param(ip, charp, 0644);
module_param(port, int, 0644);
MODULE_PARM_DESC(ip, "Server IPv4");
MODULE_PARM_DESC(port, "Server port");

static struct socket *sock = NULL;
static char *message = "Hello World! from kernel\n";
```

Network - connect_to_server

```c
static int connect_to_server(void)
{
    struct sockaddr_in server_addr = { 0 };
    int ret = 0;

    ret = sock_create(AF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
    if (ret < 0)
    {
        pr_err("network: error creating the socket: %d\n", ret);
        return 1;
    }
```

Network - connect_to_server

```c
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
ret = in4_pton(ip, -1, (u8 *)&server_addr.sin_addr.s_addr, '\0', NULL);
if (ret == 0)
{
    pr_err("network: error converting IP\n");
    return 1;
}
```

Network - connect_to_server

```
ret = sock->ops->connect(sock, (struct sockaddr *)&server_addr,\
                         sizeof(server_addr), 0);
if (ret < 0)
{
    pr_err("network: error connecting: %d\n", ret);
    sock_release(sock);
    return ret;
}

pr_info("network: Connected at %s:%d\n", ip, port);
return 0;
}
```

Network - send_message

```c
static void send_message(void)
{
struct kvec vec = { 0 };
struct msghdr msg = { 0 };
int ret = 0;

vec.iov_base = message;
vec.iov_len = strlen(message);

memset(&msg, 0, sizeof(msg));
```

Network - send_message

```
ret = kernel_sendmsg(sock, &msg, &vec, 1, vec.iov_len);
if (ret < 0)
    pr_err("network: error sending message: %d\n", ret);
else
    pr_info("network: message sended : %s", message);
}
```

Network - Test

```
# Shell 1
$ nc -lnvp 4242
...
# Shell 2
$ sudo dmesg -T -W -L
...
# Shell 3
$ sudo insmod network.ko ip="127.0.0.1" port=4242 message='"Coucou les APPING!"'
```

Be careful about how you insert the message parameter

It has to be single quotes containing double quotes

Network - What is next?

Look at some documentation about

```
struct msghdr;
struct kvec;
```

Stay connected with the server and begin to chat with it

Epidriver - Drivers

A driver is a code inside the kernel registered with a device.

We developped modules, but not exacty drivers, because we were just writing inside the kernel ring buffer.

You can check the file /proc/devices

Epidriver - Node file

Nodes are all type of file (regulars, directories, specials, sockets, pipes. . .) on the filesystem.

You can check them with the first character when long listing them.

```
$ ls -lh /dev/
$ ls -lh /dev/input/
```

Epidriver - mknod(1)

You can create a node with mknod. It needs a **major** and a **minor**.

The major and minor are integers handled by the kernel. The major is associated with a driver, and minors are separated files associated with a major.

Major will be given by your driver, the minor creation is up to you when creating the node.

You can delete a special file with rm(1) (or unlink(1)).

```
$ mknod /dev/epidriver c MAJOR MINOR
```

Epidriver - How to get the major

There are different ways. The easiest is to call register_chrdev(9). It returns the major chosen by the kernel. Then get it with /proc/devices or print it in the kernel logs.

Epidriver - EpiDriver in action

```
sudo insmod ./epidriver.ko
# check the kernel logs or cat /proc/devices to get the major
$ sudo mknod /dev/epidriver c $MAJOR 0
$ cat /dev/epidriver
...
^C
$ dd if=/dev/epidriver of=test.out bs=512 count=1 status=progress
$ cat test.out
...
$ sudo vim /dev/epidriver
    write something
    :w!
# check the kernel logs :)
$ sudo rmmod epidriver
```

Epirandom - Evolving epidriver to something more interesting

Look at the epirandom code

We are automating the insertion of a new driver with the major and minor, we do not need **mknod(1)** anymore!

Epirandom - Action!

```
$ make
...
$ sudo insmod epirandom.ko
$ cat /dev/epirandom
...
^C
$ dd if=/dev/epirandom of=test.out bs=512 count=1 status=progress
$ cat test.out
...
$ sudo rmmod epirandom
$ sudo insmod epirandom.ko alphabet='APPING'
# repeat the commands above ;)
```

?

Questions ?