Linux Kernel Programming

Lab 05: User/Kernel Communication Mechanisms

AUTHOR
Redha Gouicem and Julien Sopena

Task 1: The *sysfs* pseudo file system

The *sysfs* is a pseudo file system in the kernel that allows you to export kernel objects (structures, variables, etc.) to user space. It is usually mounted in `/sys`.

Various subsystems provide a user space API through the *sysfs*. You can find them by checking the `/sys` directory.

In previous labs, you already manipulated this API when using module parameters. The `module_param()` functions export the variables set as parameters to the *sysfs* in `/sys/module/<module name>/parameters`.

The *sysfs* architecture is tightly coupled with `struct kobject`s: for each *kobject*, there is a corresponding *sysfs* directory; and in these directories, each file corresponds to an *attribute* (`struct attribute`). Some special *kobjects* are special as they are globally defined, *e.g.*, the `kernel_kobj` variable corresponds to the `/sys/kernel` directory.

Base attributes provided by the *sysfs* (`struct attribute`) do not provide any way of manipulating files. This structure serves as a base to define more complex attributes, defining a name and an access mode.

In this exercise, we will use the `struct kobj_attribute` that allows you to define a simple attribute with read and write capabilities. This structure contains a `struct attribute` as well as operations used when the file is read from (`show`) or written to (`store`). The *sysfs* provides different macros to simplify the creation of these attributes:

- `__ATTR_RO` for read-only attributes;
- `__ATTR_WO` for write-only attributes;
- `__ATTR_RW` for read/write attributes.

When using these macros, defining a read/write attribute called `foo` will require the definition of the `foo_show()` and `foo_store()` functions too.

Once the attribute is defined, it needs to be exported as a file with the `sysfs_create_file()` function:

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
```

Conversely, deleting files in the *sysfs* is done with the `sysfs_remove_file()` function that **must** be called when unloading your modules!

In this exercise, we will always use `kernel_kobj` as a parent in order to place our files/directories in `/sys/kernel`.

Question 1

Write a *hellosysfs* module that creates the file `/sys/kernel/hello` which returns the string `"Hello sysfs!"` when read from. Does this file *really* contain the string?

To test your implementation, just read the content of the file with the `cat` command after loading your module.

Question 2

We now want to be able to write a value into `/sys/kernel/hello` in order to modify the string read later on. Modify your module to obtain this behavior:

```
$ cat /sys/kernel/hello
Hello sysfs!
$ echo -n "dude" > /sys/kernel/hello
$ cat /sys/kernel/hello
Hello dude!
```

Task 2: Introduction to ioctl

An *ioctl* is a custom system call that allows to communicate directly with the kernel. More specifically, they are used to communicate with device drivers.

From a user's perspective, the system call *ioctl* is performed on an open file, with a request number and a parameter. Have a look at the man page for more information (`man 2 ioctl`).

In the kernel, an *ioctl* is nothing more than an operation in the `struct file_operations`: `unlocked_ioctl`. The classical approach to implementing an *ioctl* is to create a device driver that implements this operation. Indeed, all devices in Linux are represented by a file (you can find them in `/dev`). When a device driver is created, adding a new device to the system consists in creating a special file (block file or character file) with the `mknod` command.

A device is identified by a major number and a name. You can observe the list of devices available in your kernel by reading the `/proc/devices` file, which contains the list of devices with their major number and name.

In the kernel, you can register a device with the following function:

```
int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops)
```

If you don't know what major number to choose, using 0 will make the kernel choose a random number and return it.

Conversely, you can delete a device by unregistering it with the following function:

```
void unregister_chrdev(unsigned int major, const char *name);
```

The `unlocked_ioctl` operation from the `struct file_operations` is executed when the `ioctl` system call is performed on the device file. It receives as parameters a request number and an `unsigned long`, and returns 0 when the *ioctl* succeeds, a negative error otherwise.

> **Caution**
>
> To communicate multiple values to the *ioctl*, the `unsigned long` parameter can be used as a pointer (`unsigned long` is always the size of a pointer in the kernel). In that case, the user space data **must** be copied to kernel space first with the `copy_from_user()` function. Similarly, when returning data from the kernel to the user, it needs to be copied to user space with the `copy_to_user()` function.

The request numbers you define are shared between the kernel (your module) and user space (the program using the *ioctl* system call), in the same fashion as system calls. In order to avoid mistakes such as sending an *ioctl* request to the wrong device, request numbers must be unique by convention. The kernel provides a set of macros to define these request numbers:

- `_IO(type, nr)`: request takes no argument;
- `_IOR(type, nr, datatype)`: request reads data from the kernel;
- `_IOW(type, nr, datatype)`: request writes data to the kernel;
- `_IOWR(type, nr, datatype)`: request reads data from and writes data to the kernel.

`type` is an 8-bit value, often a character literal, specific to a driver or subsystem (you can find the list here); `nr` is an 8-bit identifier for the command, unique for a given type; and `datatype` is the type of the variable pointed to by the argument of the *ioctl*, thus encoding the size of this type into the request number.

> **Note**
>
> In this exercise, we won't follow the types from the kernel documentation, and always use the character `'N'` as the type for the *ioctl*s we define.

The definitions of request numbers and structures shared between user and kernel space should be isolated in a separate header file that can be used both in user space (for applications using the *ioctl*) and kernel space (for the module implementing the *ioctl*).

Question 1

Implement the *helloioctl* module that creates a new character device driver named `"hello"`. For the time being, this device does not implement any operations, so you can pass an empty `struct file_operations`. You will also let the kernel choose a major number for your device.

In your init function, print the major number chosen by the kernel. And don't forget to clean up your device when unloading your module!

Load your module and check that your device has been properly registered by reading the `/proc/devices` file.

Question 2

We now want to implement a first *ioctl* on this device that returns the string `"Hello ioctl!"` to the user. To do so, you need to define a new read-only request number that you can call `HELLO`. As previously stated, use the character `'N'` as type and define it in a separate header file.

Implement the `unlocked_ioctl()` operation for your character device. You should return the string `"Hello ioctl!"` when the `HELLO` request is used, and `-ENOTTY` otherwise.

> **Important**
>
> Don't forget to use the `copy_from_user()` and `copy_to_user()` functions when moving data between user and kernel spaces!!!

To test your *ioctl*, you need to create the character device with the `mknod` command using its major number. For example, if the kernel gave you the major number 42:

```
mknod /dev/hello c 42 0
```

You will then need to write a user program that will use the `ioctl` system call on your device `/dev/hello` to get the string from it.

> **Note**
>
> You can compile your user program statically to avoid incompatibility issues with the *glibc* available in the VM image.

Task 3: Process representation in the kernel

We want to implement a module that monitors the CPU and memory usage of a process, even if it hid itself with a rootkit. To do so, we need to study two structures: `struct pid` and `struct task_struct`.

Question 1

What is the role of the `struct pid` defined in `include/linux/pid.h`?

Question 2

The members `utime` and `stime` of the `struct task_struct` record the CPU time spent in *user* and *system* mode respectively. What time unit is used here? You can have a look at how these members are used by architecture-specific code to find this information.

Task 4: Process monitoring

In this task, you will implement a module that periodically prints the CPU usage of a process in the system logs.

> **Tip**
>
> Test your module after every question!

Question 1

Create a module *taskmonitor* with a parameter `target` corresponding to the PID of the process to monitor. In this module, implement a function `int monitor_pid(pid_t pid)` that gets the `struct pid` of `target`. You can use the `find_get_pid()` function to query this structure.

Check that you correctly get the structure when the PID is valid, and handle the error properly when the PID is invalid, *i.e.* no process has this PID value.

> **⚠ Warning**
>
> The function `find_get_pid()` increments the reference counter of the `struct pid`. You **need** to put this reference when you are done with it with the function `put_pid()` !

Question 2

Define a `struct task_monitor` in your module. This structure contains one member of type `struct pid *` and will serve as a descriptor for the process being monitored.

Modify your function `monitor_pid()` so that it creates a `struct task_monitor` that will be kept alive until the module is unloaded.

When do you need to put the reference to the `struct pid` ?

Question 3

We now want to spawn a *kthread* that will periodically print the CPU usage (system and user) of the monitored process.

Write a function `int monitor_fn(void *arg)` that will be executed by a *kthread*, and that will print the CPU statistics every second if the monitored process is still alive. To do so, you will need to get the `struct task_struct` of the process with the `get_pid_task()` function. You can also query the state of a process with the `pid_alive()` function.

If the process is alive, your module should print a line like this every second:

```
        pid 128 usr 36181127855 sys 28725434471
```

In the initialization function of your module, use the `kthread_run()` function to create the *kthread* executing your function. You can use the modules we provided in the previous exercise to get some inspiration on how to use *kthreads*.

> **⚠ Warning**
>
> The function `get_pid_task()` increments the reference counter of the `struct task_struct`. Don't forget to put the reference with `put_task_struct()` when you are done with the object.

Task 5: Monitoring with the sysfs

In this task, you will modify the *taskmonitor* module to enable communication with the user space through the *sysfs* by creating the `taskmonitor` attribute (`/sys/kernel/taskmonitor`).

Question 1

In addition to the logs done by the *kthread*, we wish to be able to query the monitored process' statistics by reading the `/sys/kernel/taskmonitor` file.

Create the `taskmonitor` read-only attribute that will provide these statistics. To avoid code duplication, we recommend defining a structure `struct task_sample` to store statistics:

```
struct task_sample {
    u64 utime;
    u64 stime;
};
```

This structure can be filled with a function `get_sample()` that will return the state of the process (alive or terminated). This function can be used by your *kthread* as well as by your *sysfs* interface:

```
bool get_sample(struct task_monitor *tm, struct task_sample *sample);
```

You can test your code with the following commands (with a valid PID):

```
$ insmod taskmonitor.ko target=267
$ cat /sys/kernel/taskmonitor
pid 267 usr 9148756 sys 3834284
```

Question 2

We now want to be able to suspend the *kthread* execution without unloading the module. We can do this by enabling write operations to our *sysfs* attribute `taskmonitor`. When a user writes `"stop"` into the *sysfs* file, the *kthread* should be stopped and destroyed. When a user writes `"start"`, a new *kthread* is started. When the *kthread* is stopped, users should still be able to query the statistics by reading the *sysfs* file.

```
[  383.520546] pid 132 usr 210079440116 sys 161863404610
[  384.544370] pid 132 usr 210644137037 sys 162321252182
$ echo -n stop > /sys/kernel/taskmonitor
[  384.885695] halting monitor thread
$ cat /sys/kernel/taskmonitor
pid 132 usr 216043870064 sys 166419115138
$ cat /sys/kernel/taskmonitor
pid 132 usr 216917620026 sys 167117974450
$ echo -n start > /sys/kernel/taskmonitor
[  399.237499] starting monitor thread
[  399.241370] pid 132 usr 218937548440 sys 168714926613
[  400.288363] pid 132 usr 219525535237 sys 169173913270
```

Modify the *taskmonitor* module to implement these new features. You will be careful to **never** have multiple *kthreads* running at the same time.

Task 6: Monitoring with ioctl

In this task, you will implement a similar mechanism as with the *sysfs* previously, but using *ioctl*s.

Question 1

Register your task monitor as a character device driver that will be used to control your module. Again, you can let the kernel pick a major number for you.

Question 2

In addition to the *kthread* logs and the *sysfs* interface, we wish to be able to query the statistics of the monitored process with an *ioctl*.

Add the request number `TM_GET` to perform a statistics query. There are two ways to implement this *ioctl*, and you may try both:

- pass a buffer as a parameter and fill it with the string containing the statistics;
- pass a pointer to a `struct task_sample` that will be filled by the module, and let the user program use it however it wants to. With this approach, you should define the request number and the structure passed as an argument in a header file shared by both your module and your user applications.

To test your code, write a small program that queries samples and displays them periodically.

Question 3

Let's now implement the second feature available in our *sysfs* interface: controlling the *kthread* without unloading the module. To do so, you will add two new requests without parameters:

- `TM_STOP` stops the *kthread* if it is running;
- `TM_START` starts the *kthread* if it is halted.

Again, you will make sure that there is always at most **one** *kthread* running on the system.

Write a test program that manipulates these new *ioctl* requests.

Question 4

We now want to be able to get and set the PID monitored by our module without (un)loading it.

Add a new read-write *ioctl* request to your module, `TM_PID`, with a parameter of type `pid_t`:

- if the parameter is negative, the request does not set a new PID, but just gets the current value back;
- if the parameter is positive, the request changes the monitored process to the one with the PID passed.

Don’t forget to properly handle errors, *e.g.*, the PID requested does not exist.