Want to play a game?

Johan Montelius

HT2018

Introduction

This is a simple exercise to compile and insert a small program into the kernel i.e. a kernel module. The program that you will write doesn't do anything useful but you will learn the basics steps of writing a kernel module.

You will need *root privileges* on the machine that you are working on. The safest way is of course if you run everything using a virtual machine but you could of course use your regular machine (at your own risk). You will not be able to use the KTH computers since you don't have root access to those and will (at least as it is now) not be able to start up a virtual machine on those.

The kernel is of course the heart of the operating system and has complete access to the hardware that it is running on. User space processes are protected from each other but the kernel has complete access to everything. If we do a mistake in our user space, a process will crash but the kernel survives - a mistake in the kernel will most often result in that the machine crashes (but you will most often be able to restart it).

There are basically two ways of extending the kernel, either we compile our own modified kernel or we insert a module into a running kernel. The latter approach is of course much more convenient but it requires that we know what we're doing (or at least have an idea of what we're doing).

1 A first try

The program that we will write is a regular C program but with some special properties. It does not contain a main procedure, instead it will have several procedures that will be used by the kernel, and could be made available to user level programs.

The two most important procedures are the ones used when the module is loaded and removed from the kernel. Save the following in a file called `hal.c`.

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dr Chandra");
```

```
MODULE_DESCRIPTION("Heuristically programmed ALgorithmic computer");

static int __init hal_init(void)
{
    printk(KERN_INFO "I honestly think you ought to calm down;\n");
    return 0;
}

static void __exit hal_cleanup(void)
{
    printk(KERN_INFO "What are you doing, Dave?\n");
}

module_init(hal_init);
module_exit(hal_cleanup);
```

In this program we first include some header files that we need. We then describe our kernel module and it is important that we set thelicense to "GPL" i.e. that this is *free software* and not some proprietary code. We also provide two straneg looking functions that we use in the two last macros: module_init() and module_exit(). Our kernel module will do nothing but write out a message when it is inserted into the kernel and when it is removed from the kernel.

We should now compile this but we need to compile it using some special libraries that are used when the kernel is built. If you look in the directory `/lib/modules`, you will find a directory for each linux kernel that you have installed on your computer (or virtual machine). In these directories you will find a link called `build` that is referring to the source directory of the kernel.

1.1 a Makefile

The source directory has a `Makefile` that will do the compilation for us. To make things easy, we create our own makefile in our source directory that will do the making for us. Create a file called `Makefile` in the directory that holds the file `hal.c`. The trick with the `uname -r` will get us the number of the running kernel and thus direct us to the right directory.

```
obj-m += hal.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

Now if everything works you should be able to *make* the module from the directory of the makefile.

```
$ make
  :
  :
make[1]: Leaving directory '/usr/src/linux-headers-4.4.0-34-generic'
```

Now take a look in the directory and you will find tons of new files. If you use the command `ls -a` you will see even more. The one that we are interested in is the file `hal.ko`, this is the kernel module object file.

1.2 loading the module

So we have an object file and the only thing now is to load it into the kernel. You can take a look at all the modules that are already loaded into the kernel by using the `lsmod` command.

```
$ lsmod
  :
  :
```

Now if you have the right privilege you should be able to insert also our module. Try this:

```
$ sudo insmod hal.ko
```

If nothing happens you have probably succeeded. Now issue the command `dmesg` that will print a file containing all messages from the kernel since you booted your machine. Pipe the output to `tail` to see the last rows. The last entry is hopefully a message from our HAL module - it's now in control of your machine.

```
$ dmesg | tail
  :
  :
```

You can also verify, using `lsmod`, that the module is in fact among the loaded modules. To remove it from the kernel we use the command `rmmod` giving the name of the module as an argument.

```
$ sudo rmmod hal
```

To show you that the module is actually loaded in to kernel space we can check the address it is loaded to. Change the print-out when the module is loaded to the following, make, load the module and check the message from `dmesg`. Is it an address in kernel space?

```
here:
  printk(KERN_INFO "I'm here %p);\n", &&here);
```

2 Now what

This assignment is only scratching on the surface of kernel space programming. Moving forward from here things require a lot more coding; it's not complicated but it's more code. One would like to simply add a procedure in the loaded kernel module and then make this procedure accessible as a system call. This would make things easy for the programmer but probably result in a bowl of spaghetti once everyone added their own system calls. The number of true system calls should be kept small to make the operating system easier to manage and control.

The alternative way of communicating with our kernel module is to let it show up with a regular file interface. To the user level programs it looks like a file but under the hood the module is doing the work.

2.1 Dr Dyson to your help

Create a new directory called skynet and copy the makefile we used to the new directory. Edit the Makefile so that it now used an object file called "skynet.o". Create a file called `skynet.c` with the following code; we will go through the code so that you know why it is there.

The header files are as before but we are now going to add a file system interface to the module. We therefore include two more files `proc_fs.h` and `seq_file.h`. The macros that describe the module are required so we write some fun things.

```
#include <linux/module.h>     // included for all kernel modules
#include <linux/kernel.h>     // included for KERN_INFO
#include <linux/init.h>       // included for __init and __exit macros

#include <linux/proc_fs.h>    // file operations
#include <linux/seq_file.h>   // seq_read, ...

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dr. Dyson");
MODULE_DESCRIPTION("Global Information Grid");
```

Next, we are defining a `proc_ops` data structure. This structure is populated with call-back functions that the kernel will use when the file interface of the module is used. The kernel needs to know how the file should be opened, closed, read from etc. The only function that we will provide is the function used to open the file `skynet_open()`.

```
static int skynet_show(struct seq_file *m, void *v);

static int skynet_open(struct inode *inode, struct  file *file);

```

```c
static const struct proc_ops skynet_fops = {
     .proc_open = skynet_open,
     .proc_read = seq_read,
     .proc_lseek = seq_lseek,
     .proc_release = single_release,
};
```

```c
static int skynet_show(struct seq_file *m, void *v) {
 here:
  seq_printf(m, "Skynet location: 0x%lx\n",
                  (unsigned long)&&here);
  return 0;
}

static int skynet_open(struct inode *inode, struct  file *file) {
  return single_open(file, skynet_show, NULL);
}
```

We then provide the functions that will be used when the module is loaded and unloaded. This time we will actually do some work here. We will register the module as a `proc` module and provide a name ‘‘`skynet`’’ that will show up the the `/proc` directory. Wen the module is unloaded we remove the proc-entry.

```c
static int __init skynet_init(void) {
  proc_create("skynet", 0, NULL, &skynet_fops);
  printk(KERN_INFO "Skynet in control\n");

  return 0;
}

static void __exit skynet_cleanup(void) {
  remove_proc_entry("skynet", NULL);
  printk(KERN_INFO "I'll be back!\n");
}

module_init(skynet_init);
module_exit(skynet_cleanup);
```

2.2 in control

Make the module and load it into the kernel, check the output from `dmesg` to make sure that it was loaded ok. Then take a look in the `/proc` directory, do we have a `skynet` file? Take a closer look at the file using `ls -l`, what does it look like, how big is it?

```
$ ls -l /proc/skynet
  :
```

Now try to read the file using the `cat` command.

```
$ cat /proc/skynet
  :
```

This is how all the files you see in the `/proc` directory work, they are simply interfaces to different kernel services.

3 Device drivers

Take a look in the directory `/dev`, all of the files that you find there are also interfaces to kernel services. If you use the command `ls -l` you will see that the first letter of each description is either `d`, `l` or something that you might not have seen before: `b` or `c`. The latter files are *block* and *character devices*. We will now try to add a kernel module that shows up as a character device.

3.1 Joshua

There will be some code but you will manage. Create a new directory called *joshua*, and in there a new source code file `joshua.c` and a header file `joshua.h`. Also make a copy of the Makefile and place it in the directory. Edit the Makefile so that it will use the object file `joshua.o`. Now for the code, we will start with the header file.

```
#ifndef JOSHUA
#define JOSHUA
#include <linux/ioctl.h>

// This is the required size of the buffer.

#define JOSHUA_MAX 40

// This macro will give us the right ioctl code.

#define JOSHUA_GET_QUOTE _IOR(0xff, 1, char *)

#endif
```

We will create a kernel module that will return quotes by Joshua. We have a header file that defines two macros `JOSHUA_MAX` and `JOSHUA_GET_QUOTE`.

The first is the longest string that a quota can be (including trailing zero). The user level program should allocate a buffer with a least this size and pass a pointer to the buffer to the joshua module. The joshua module will then copy a new quote from Joshua into the buffer.

The second macro is a bit cryptic but it uses a macro from the `ioctl.h` header file that helps us create a hopefully unique *ioctl number*. The number is created from one *magic number* (`0xff`), a number that is unique for the joshua module and the data type that we will pass from the user to the kernel module. We have here chosen `0xff` as our magic number but some more care should go in to this if we actually did something serious.

3.2 the user program

The user program that should request quotes from joshua could look like follows. Create a source file called `quote.c` in the same directory as joshua (not required but we need to find the joshua header file so we might as well place it there).

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>


#include "joshua.h"

int main()  {
  char *file_name = "/dev/joshua";
  int fd;

  fd = open(file_name, O_RDONLY);

  if (fd == -1) {
    perror("Joshua is not available");
    return 2;
  }

  char buffer[JOSHUA_MAX];

  if (ioctl(fd, JOSHUA_GET_QUOTE, &buffer) == -1) {
    perror("Hmm, not so good");
  } else {
    printf("Quote - %s\n", buffer);
  }
```

```
  close (fd);
  return 0;

}
```

The user program opens a file, `/dev/joshua`, in read mode and then makes a system call `ioctl()` that will make the magical request to the joshua module. Note that nothing is working so far, we have not created our module nor made it accessible from the device file. This is just to get an idea of how things will work in the end.

As you see the program creates a buffer on the stack and passes the address of this buffer to the system call. This is why we said that the data type was `char*` when we defined the ioctl number.

3.3 the module

So now we are ready to define our kernel module `joshua.c`. This will have the same components as the hal module that we created before but will also register itself using the ioctl functionality. We will go through the code part by part and explain why it is there.

The first part is a sequence of included header files. Some you have seen from `hal.c` and `skynet.c` but many are new. It's not important to keep track of which files are needed since this is quite easily determined. We also include our own `joshua.h` so we use the same size of the buffer and the same ioctl number as the user space program.

```
#include <linux/module.h>     // all kernel modules
#include <linux/kernel.h>     // KERN_INFO

#include <linux/fs.h>         // file_operations ..

#include <linux/cdev.h>       // cdev_init, cdev_all ...
#include <linux/device.h>     // class_create ...
#include <linux/uaccess.h>    // copy_to_user

#include "joshua.h"           // to agree on the interface
```

As before we need to describe the module.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Prof Franken");
MODULE_DESCRIPTION("Joshua");
```

We then define some macros and data structures that are needed when we define our device driver. The important thing here is the file operations data structure `joshua_fops`. As before we populate it with the functions that the kernel needs but now we include a function called `joshua_ioctl` that will be the interface to the device driver.

```
#define FIRST_MINOR 0
#define MINOR_CNT 1

static int joshua_open(struct inode *i, struct file *f);
static int joshua_close(struct inode *i, struct file *f);
static long joshua_ioctl(struct file *f,
                         unsigned int cmd, unsigned long arg);

static dev_t dev;
static struct cdev c_dev;
static struct class *cl;

static struct file_operations joshua_fops = {
  .owner = THIS_MODULE,
  .open = joshua_open,
  .release = joshua_close,
  .unlocked_ioctl = joshua_ioctl
};
```

Next follows declarations of our own data structures. We define an array of three quotes and an integer that we will increment to keep track of the next quote to deliver.

```
#define QUOTES 3

static const char * quotes[QUOTES] = {
  "Why play a game that cannot be won?",
  "Mutual destruction is not a victory.",
  "Simulation is the mother of knowledge."
};

static int next = 0;
```

Now for the basic definitions of the kernel module i.e. what should be done when the module is loaded and unloaded. This is of course where we want to register the module under a device name i.e. `/dev/joshua`. The first one is the procedure when the module is loaded. It looks quite scary but most of it is code to handle things that could go wrong. If you go through the code you will see that it basically, does five things:

- `alloc_chrdev_region()`: allocates a device number `dev`
- `cdev_init()`: initializes the character device structure `c_dev`

- `cdev_add()`: adds the character device given the device number
- `class_create()`: creation of a device class called `char`
- `device_create()`: creating the device given the class, device number an name "joshua"

Don't ask me how things work, the important thing is that it is doable. We can worry about what actually is happening later. Notice that most of the code is checking if the operations were successful and if not undoing the previous operations before returning an error message.

```
static int __init joshua_init(void) {
  int ret;
  struct device *dev_ret;

  printk(KERN_INFO "Want to play a game?\n");

  if ((ret = alloc_chrdev_region(&dev, FIRST_MINOR, MINOR_CNT, "joshua")) < 0) {
    return ret;
  }

  cdev_init(&c_dev, &joshua_fops);

  if ((ret = cdev_add(&c_dev, dev, MINOR_CNT)) < 0) {
    return ret;
  }

  if (IS_ERR(cl = class_create(THIS_MODULE, "char")))  {
    cdev_del(&c_dev);
    unregister_chrdev_region(dev, MINOR_CNT);
    return PTR_ERR(cl);
  }
  if (IS_ERR(dev_ret = device_create(cl, NULL, dev, NULL, "joshua"))) {
    class_destroy(cl);
    cdev_del(&c_dev);
    unregister_chrdev_region(dev, MINOR_CNT);
    return PTR_ERR(dev_ret);
  }

  return 0;
}
```

When the module is unloaded we must remove everything that we have created. We destroy the device, the class and the cdev structure - basically, undoing everything we did when the module is loaded. We also add a print statement so we know that the module was successfully unloaded.

```
static void __exit joshua_exit(void) {
  device_destroy(cl, dev);
  class_destroy(cl);
  cdev_del(&c_dev);
  unregister_chrdev_region(dev, MINOR_CNT);

  printk(KERN_INFO "How about a nice game of chess?\n");
}
```

We will now define what should be done when someone opens or closes the "file". For the user level program the device looks like a file and the obvious operations are thus open, close etc. As you see below, we don't do anything special when someone tries to open or close the file but here is where we could initialize or deleted data structures that pertains to the session.

```
static int joshua_open(struct inode *i, struct file *f) {
  return 0;
}

static int joshua_close(struct inode *i, struct file *f) {
  return 0;
}
```

Now for the heart of our module, the things we will do when someone issues a `ioctl()` command. This is where we will return a quote from Joshua. Remember that the client allocated a buffer and sent us a address to this buffer. It's our job to copy one of the quotes that we have into this buffer.

As you see below the ioctl procedure takes three arguments: a file descriptor, a *command* and the argument that was passed from the user. The command is a number that has been generated by the `_IOR` macro. The user program implicitly used this when using the `JOSHUA_GET_QUOTE` macro in the `joshua.h` header file. We can now use the same macro and have a switch statement that looks at `cmd` and hopefully jumps to the right case. You now see how we very easily can add new commands to our module.

```
static long joshua_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{

  switch (cmd) {

    case JOSHUA_GET_QUOTE:

      next = (next+1) % QUOTES;
```

```
      printk(KERN_INFO "Joshua: copy to buffer at 0x%lx\n", arg);

      if (copy_to_user((char *)arg, quotes[next], JOSHUA_MAX))
        {
          return -EACCES;
        }
      break;

    default:
      return -EINVAL;
    }
  return 0;
}
```

To handle a `JOSHUA_GET_QUOTE` request, we increment the `next` variable, print a message and copy the next quote into the buffer provided to us in `arg`. Note that this is a very scary procedure in that we don't really know what the user sent us. In the best case it is actually a memory reference to a buffer that is at least `JOSHUA_MAX` large. If the user made mistake, the buffer is too small or the address is pointing randomly somewhere else. In the worst case the user is luring us into writing something to a memory segment that belongs to the kernel. The user has of course no possibility to write to these segments but the joshua module belongs to the kernel and has complete access to the kernel space. This is why we use the procedure `copy_to_user()` that will check that the address actually belongs to the user address space and only then copy the string.

The last thing we do is use the macros `module_init()` and `module_exit()` to register our procedures that should be used when the module is loaded and unloaded.

```
module_init(joshua_init);
module_exit(joshua_exit);
```

This is it, you should be set to go.

4 Want to play a game

Compile the joshua module using the make file in the joshua directory. Then load the module using `insmod`.

```
$ make
  :
  :
$ sudo insmod joshua.ko
```

Now take a look in the `/dev` and you should see a `joshua` device. Do `ls -l` and you will have more information.

```
 $ ls -l /dev/joshua
crw------- 1 root root 244, 0 aug 19 10:16 /dev/joshua
```

Hmm, a `character device` (the 'c' in the beginning) with read and write privileges for `root`, who is also the owner of this file. To be able to open it from a regular user we need to change the privilege level.

```
 $ sudo chmod o+r /dev/joshua
```

Do `ls -l` again and see that the mode of the device now allows "other" users to read the device. Also take a look in `/sys/devices/virtual/char` and you will see a directory containing information about our device. We don't really need to know what is going on here but I'm quite sure there is some magic involved. The important thing is that we have our device available for the users.

Now switch attention to the program `quote.c`, compile it and run a test, does it work?

5 Summary

This assignment is of course only scratching the surface of kernel modules or device driver implementation. The important lesson is that it is fairly easy to add things to the kernel and that user level programs can interact with kernel module using the device interface.

You're encouraged to play around some more, add more commands or ponder what would happen if two processes call the module at the same time. When you experiment, keep in mind that you're playing with the kernel; hopefully on a virtual machine but if you're like me you of course run it on your own laptop where you also have all you non-backed photos. Another thing you must know is that most of the libraries that you are used to use are not available to the kernel. The libraries are written to be called from user space and maybe trap to the kernel for a system call - but we are already in the kernel. This is why the code above have used `printk` for output.

Appendix

Hear are some comments on things that could go wrong and how to fix them:

things do not work

Do you have “make” installed? If not install it using “sudo apt install make”. You will use make in more assignments so it’s a good thing to have anyway.

You need to have the library development files i.e. the header files that we refer to. If things go wrong when you make the modules this could be the problem. Install them using “sudo apt install libelf-dev”

Make sure that the make file is actually called “Makefile”. When you run make it will look for this file in the current directory. You could change this using the “-f” flag but we will only use one Makefile for each experiment and each experiment has its own directory.

The Makefile must of course be adapted for the different experiments. If the first Makefile is used for “hal.o” then you will have to change this to for example “skynet.o”.

what is going on

The Makefile defines a variable “obj-m += hal.o” (or rather adds “hal.o” to the “obj-m” variable. This variable is used by the Makefile in the “
lib
modules
..” directory. It will be the object file that is turned in to a kernel object file. When we run make, it will read our Makefile and then, if no arguments are given, perform what is listed under “all:”. What we say there is that it should run make but now with some arguments.

The first argument to make is the “-C” flag together with a mysteriously looking directory. The flag will tell make to go to this directory and then do what ever needs to be done there. On my current machine the directory is:

```
/lib/modules/4.15.0-36-generic/build
```

but this will of course change when ever I upgrade my system. To avoid having to change the Makefile every time I use the trick of calling the command “uname”. Try the command “uname -a” in a shell and you will see a lot of information about your system. The thing that we are looking for is the kernel revision that we get from “uname -r”.

In the same way we define the variable “M” to be equal to the directory where we are currently standing. This is where the source files are found and this is were the results should be written. We also give the argument “modules” since we want make to generate the kernel modules for us.