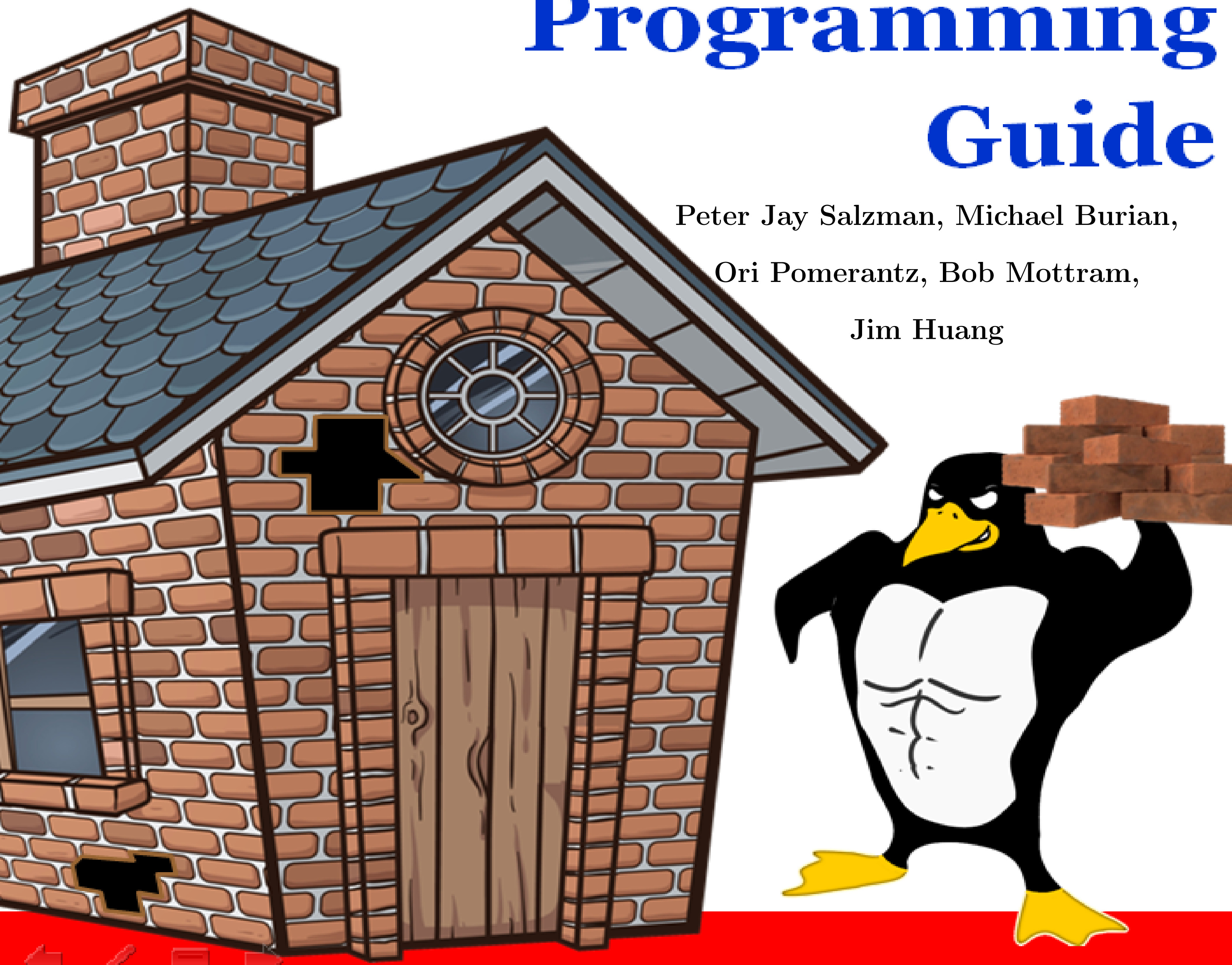
The Linux Kernel Module Programming Guide

Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, Jim Huang

August 12, 2024

The Linux Kernel Module Programming Guide

Peter Jay Salzman, Michael Burian,
Ori Pomerantz, Bob Mottram,
Jim Huang



Contents

1 Introduction

The Linux Kernel Module Programming Guide is a free book; you may reproduce and/or modify it under the terms of the Open Software License, version 3.0.

This book is distributed in the hope that it would be useful, but without any warranty, without even the implied warranty of merchantability or fitness for a particular purpose.

The author encourages wide distribution of this book for personal or commercial use, provided the above copyright notice remains intact and the method adheres to the provisions of the Open Software License. In summary, you may copy and distribute this book free of charge or for a profit. No explicit permission is required from the author for reproduction of this book in any medium, physical or electronic.

Derivative works and translations of this document must be placed under the Open Software License, and the original copyright notice must remain intact. If you have contributed new material to this book, you must make the material and source code available for your revisions. Please make revisions and updates available directly to the document maintainer, Jim Huang <jserv@ccns.ncku.edu.tw>. This will allow for the merging of updates and provide consistent revisions to the Linux community.

If you publish or distribute this book commercially, donations, royalties, and/or printed copies are greatly appreciated by the author and the Linux Documentation Project (LDP). Contributing in this way shows your support for free software and the LDP. If you have questions or comments, please contact the address above.

1.1 Authorship

The Linux Kernel Module Programming Guide was initially authored by Ori Pomerantz for Linux v2.2. As the Linux kernel evolved, Ori's availability to maintain the document diminished. Consequently, Peter Jay Salzman assumed the role of maintainer and updated the guide for Linux v2.4. Similar constraints arose for Peter when tracking developments in Linux v2.6, leading to Michael Burian joining as a co-maintainer to bring the guide up to speed with Linux v2.6. Bob Mottram contributed to the guide by updating examples for Linux v3.8 and later. Jim Huang then undertook the task of updating the guide for recent Linux versions (v5.0 and beyond), along with revising the LaTeX document.

1.2 Acknowledgements

The following people have contributed corrections or good suggestions:

Amit Dhingra, Andy Shevchenko, Arush Sharma, Benno Bielmeier, Bob Lee, Brad Baker, Che-Chia Chang, Cheng-Shian Yeh, Chih-En Lin, Chih-Hsuan Yang, Chih-Yu Chen, Ching-Hua (Vivian) Lin, Chin Yik Ming, cvvletter,

Cyril Brulebois, Daniele Paolo Scarpazza, David Porter, demonsome, Dimo Velev, Ekang Monyet, Ethan Chan, Francois Audeon, Gilad Reti, heartofrain, Horst Schirmeier, Hsin-Hsiang Peng, Ignacio Martin, I-Hsin Cheng, Iûnn Kiàn-îng, Jian-Xing Wu, Johan Calle, keytouch, Kohei Otsuka, Kuan-Wei Chiu, manbing, Marconi Jiang, mengxinayan, Meng-Zong Tsai, Peter Lin, Roman Lakeev, Sam Erickson, Shao-Tse Hung, Shih-Sheng Yang, Stacy Prowell, Steven Lung, Tristan Lelong, Tse-Wei Lin, Tucker Polomik, Tyler Fanelli, VxTeemo, Wei-Hsin Yeh, Wei-Lun Tsai, Xatierlike Lee, Yen-Yu Chen, Yin-Chiuan Chen, Yi-Wei Lin, Yo-Jung Lin, Yu-Hsiang Tseng, YYGO.

1.3 What Is A Kernel Module?

Involvement in the development of Linux kernel modules requires a foundation in the C programming language and a track record of creating conventional programs intended for process execution. This pursuit delves into a domain where an unregulated pointer, if disregarded, may potentially trigger the total elimination of an entire file system, resulting in a scenario that necessitates a complete system reboot.

A Linux kernel module is precisely defined as a code segment capable of dynamic loading and unloading within the kernel as needed. These modules enhance kernel capabilities without necessitating a system reboot. A notable example is seen in the device driver module, which facilitates kernel interaction with hardware components linked to the system. In the absence of modules, the prevailing approach leans toward monolithic kernels, requiring direct integration of new functionalities into the kernel image. This approach leads to larger kernels and necessitates kernel rebuilding and subsequent system rebooting when new functionalities are desired.

1.4 Kernel module package

Linux distributions provide the commands `modprobe`, `insmod` and `depmod` within a package.

On Ubuntu/Debian GNU/Linux:

```
sudo apt-get install build-essential kmod
```

On Arch Linux:

```
sudo pacman -S gcc kmod
```

1.5 What Modules are in my Kernel?

To discover what modules are already loaded within your current kernel use the command `lsmod`.

```
sudo lsmod
```

Modules are stored within the file `/proc/modules`, so you can also see them with:

```
sudo cat /proc/modules
```

This can be a long list, and you might prefer to search for something particular. To search for the `fat` module:

```
sudo lsmod | grep fat
```

1.6 Is there a need to download and compile the kernel?

To effectively follow this guide, there is no obligatory requirement for performing such actions. Nonetheless, a prudent approach involves executing the examples within a test distribution on a virtual machine, thus mitigating any potential risk of disrupting the system.

1.7 Before We Begin

Before delving into code, certain matters require attention. Variances exist among individuals' systems, and distinct personal approaches are evident. The achievement of successful compilation and loading of the inaugural "hello world" program may, at times, present challenges. It is reassuring to note that overcoming the initial obstacle in the first attempt paves the way for subsequent endeavors to proceed seamlessly.

1. Modversioning. A module compiled for one kernel will not load if a different kernel is booted, unless `CONFIG_MODVERSIONS` is enabled in the kernel. Module versioning will be discussed later in this guide. Until module versioning is covered, the examples in this guide may not work correctly if running a kernel with modversioning turned on. However, most stock Linux distribution kernels come with modversioning enabled. If difficulties arise when loading the modules due to versioning errors, consider compiling a kernel with modversioning turned off.

2. Using X Window System. It is highly recommended to extract, compile, and load all the examples discussed in this guide from a console. Working on these tasks within the X Window System is discouraged.

   Modules cannot directly print to the screen like `printf()` can, but they can log information and warnings that are eventually displayed on the screen, specifically within a console. If a module is loaded from an `xterm`, the information and warnings will be logged, but solely within the systemd journal. These logs will not be visible unless consulting the `journalctl`. Refer to 4 for more information. For instant access to this information, it is advisable to perform all tasks from the console.

3. SecureBoot. Numerous modern computers arrive pre-configured with UEFI SecureBoot enabled—an essential security standard ensuring booting exclusively through trusted software endorsed by the original equipment manufacturer. Certain Linux distributions even ship with the default Linux kernel configured to support SecureBoot. In these cases, the kernel module necessitates a signed security key.

   Failing this, an attempt to insert your first "hello world" module would result in the message: "*ERROR: could not insert module*". If this message *Lockdown: insmod: unsigned module loading is restricted; see man kernel lockdown.7* appears in the `dmesg` output, the simplest approach involves disabling UEFI SecureBoot from the boot menu of your PC or laptop, allowing the successful insertion of "hello world" module. Naturally, an alternative involves undergoing intricate procedures such as generating keys, system key installation, and module signing to achieve functionality. However, this intricate process is less appropriate for beginners. If interested, more detailed steps for SecureBoot can be explored and followed.

2 Headers

Before building anything, it is necessary to install the header files for the kernel.

On Ubuntu/Debian GNU/Linux:

```
sudo apt-get update
apt-cache search linux-headers-`uname -r`
```

The following command provides information on the available kernel header files. Then for example:

```
sudo apt-get install kmod linux-headers-5.4.0-80-generic
```

On Arch Linux:

```
sudo pacman -S linux-headers
```

On Fedora:

```
sudo dnf install kernel-devel kernel-headers
```

3 Examples

All the examples from this document are available within the `examples` subdirectory.

Should compile errors occur, it may be due to a more recent kernel version being in use, or there might be a need to install the corresponding kernel header files.

4 Hello World

4.1 The Simplest Module

Most individuals beginning their programming journey typically start with some variant of a *hello world* example. It is unclear what the outcomes are for those who deviate from this tradition, but it seems prudent to adhere to it. The learning process will begin with a series of hello world programs that illustrate various fundamental aspects of writing a kernel module.

Presented next is the simplest possible module.

Make a test directory:

```
mkdir -p ~/develop/kernel/hello-1
cd ~/develop/kernel/hello-1
```

Paste this into your favorite editor and save it as `hello-1.c`:

```c
/*
 * hello-1.c - The simplest kernel module.
 */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

int init_module(void)
{
    pr_info("Hello world 1.\n");

    /* A non 0 return means init_module failed; module can't be loaded. */
    return 0;
}

void cleanup_module(void)
{
    pr_info("Goodbye world 1.\n");
}

MODULE_LICENSE("GPL");
```

Now you will need a `Makefile`. If you copy and paste this, change the indentation to use *tabs*, not spaces.

```
obj-m += hello-1.o

PWD := $(CURDIR)

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

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

In `Makefile`, `$(CURDIR)` can set to the absolute pathname of the current working directory(after all `-C` options are processed, if any). See more about `CURDIR` in GNU make manual.

And finally, just run `make` directly.

```
make
```

If there is no `PWD := $(CURDIR)` statement in Makefile, then it may not compile correctly with `sudo make`. Because some environment variables are specified by the security policy, they can't be inherited. The default security policy is `sudoers`. In the `sudoers` security policy, `env_reset` is enabled by default, which restricts environment variables. Specifically, path variables are not retained from the user environment, they are set to default values (For more information see: sudoers manual). You can see the environment variable settings by:

```
$ sudo -s
# sudo -V
```

Here is a simple Makefile as an example to demonstrate the problem mentioned above.

```
all:
	echo $(PWD)
```

Then, we can use `-p` flag to print out the environment variable values from the Makefile.

```
$ make -p | grep PWD
PWD = /home/ubuntu/temp
OLDPWD = /home/ubuntu
echo $(PWD)
```

The `PWD` variable won't be inherited with `sudo`.

```
$ sudo make -p | grep PWD
echo $(PWD)
```

However, there are three ways to solve this problem.

1. You can use the `-E` flag to temporarily preserve them.

```
$ sudo -E make -p | grep PWD
PWD = /home/ubuntu/temp
OLDPWD = /home/ubuntu
echo $(PWD)
```

2. You can set the `env_reset` disabled by editing the `/etc/sudoers` with root and `visudo`.

```
## sudoers file.
##
...
Defaults env_reset
## Change env_reset to !env_reset in previous line to keep all
↪  environment variables
```

Then execute `env` and `sudo env` individually.

```
# disable the env_reset
echo "user:" > non-env_reset.log; env >>
↪  non-env_reset.log
echo "root:" >> non-env_reset.log; sudo env >>
↪  non-env_reset.log
# enable the env_reset
echo "user:" > env_reset.log; env >> env_reset.log
echo "root:" >> env_reset.log; sudo env >>
↪  env_reset.log
```

You can view and compare these logs to find differences between `env_reset` and `!env_reset`.

3. You can preserve environment variables by appending them to `env_keep` in `/etc/sudoers`.

```
Defaults env_keep += "PWD"
```

After applying the above change, you can check the environment variable settings by:

```
$ sudo -s
# sudo -V
```

If all goes smoothly you should then find that you have a compiled `hello-1.ko` module. You can find info on it with the command:

```
modinfo hello-1.ko
```

At this point the command:

```
sudo lsmod | grep hello
```

should return nothing. You can try loading your shiny new module with:

```
sudo insmod hello-1.ko
```

The dash character will get converted to an underscore, so when you again try:

```
sudo lsmod | grep hello
```

You should now see your loaded module. It can be removed again with:

```
sudo rmmod hello_1
```

Notice that the dash was replaced by an underscore. To see what just happened in the logs:

```
sudo journalctl --since "1 hour ago" | grep kernel
```

You now know the basics of creating, compiling, installing and removing modules. Now for more of a description of how this module works.

Kernel modules must have at least two functions: a "start" (initialization) function called `init_module()` which is called when the module is `insmod`ed into the kernel, and an "end" (cleanup) function called `cleanup_module()` which is called just before it is removed from the kernel. Actually, things have changed starting with kernel 2.3.13. You can now use whatever name you like for the start and end functions of a module, and you will learn how to do this in Section 4.2.

In fact, the new method is the preferred method. However, many people still use `init_module()` and `cleanup_module()` for their start and end functions.

Typically, `init_module()` either registers a handler for something with the kernel, or it replaces one of the kernel functions with its own code (usually code to do something and then call the original function). The `cleanup_module()` function is supposed to undo whatever `init_module()` did, so the module can be unloaded safely.

Lastly, every kernel module needs to include `<linux/module.h>`. We needed to include `<linux/printk.h>` only for the macro expansion for the `pr_alert()` log level, which you'll learn about in Section 2.

1. A point about coding style. Another thing which may not be immediately obvious to anyone getting started with kernel programming is that indentation within your code should be using **tabs** and **not spaces**. It is one of the coding conventions of the kernel. You may not like it, but you'll need to get used to it if you ever submit a patch upstream.

2. Introducing print macros. In the beginning there was `printk`, usually followed by a priority such as `KERN_INFO` or `KERN_DEBUG`. More recently this can also be expressed in abbreviated form using a set of print macros, such as `pr_info` and `pr_debug`. This just saves some mindless keyboard bashing and looks a bit neater. They can be found within include/linux/printk.h. Take time to read through the available priority macros.

3. About Compiling. Kernel modules need to be compiled a bit differently from regular userspace apps. Former kernel versions required us to care much about these settings, which are usually stored in Makefiles. Although hierarchically organized, many redundant settings accumulated in sublevel Makefiles and made them large and rather difficult to maintain. Fortunately, there is a new way of doing these things, called kbuild, and the build process for external loadable modules is now fully integrated into the standard kernel build mechanism. To learn more on how to compile modules which are not part of the official kernel (such as all the examples you will find in this guide), see file Documentation/kbuild/modules.rst.

   Additional details about Makefiles for kernel modules are available in Documentation/kbuild/makefiles.rst. Be sure to read this and the related files before starting to hack Makefiles. It will probably save you lots of work.

> Here is another exercise for the reader. See that comment above the return statement in `init_module()`? Change the return value to something negative, recompile and load the module again. What happens?

4.2 Hello and Goodbye

In early kernel versions you had to use the `init_module` and `cleanup_module` functions, as in the first hello world example, but these days you can name those anything you want by using the `module_init` and `module_exit` macros. These macros are defined in include/linux/module.h. The only requirement is that your init and cleanup functions must be defined before calling the those macros, otherwise you'll get compilation errors. Here is an example of this technique:

```c
/*
 * hello-2.c - Demonstrating the module_init() and module_exit() macros.
 * This is preferred over using init_module() and cleanup_module().
 */
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int __init hello_2_init(void)
{
    pr_info("Hello, world 2\n");
    return 0;
}

static void __exit hello_2_exit(void)
{
    pr_info("Goodbye, world 2\n");
}

module_init(hello_2_init);
module_exit(hello_2_exit);

MODULE_LICENSE("GPL");
```

So now we have two real kernel modules under our belt. Adding another module is as simple as this:

```makefile
obj-m += hello-1.o
obj-m += hello-2.o

PWD := $(CURDIR)

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

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

Now have a look at drivers/char/Makefile for a real world example. As you can see, some things got hardwired into the kernel (`obj-y`) but where have all those `obj-m` gone? Those familiar with shell scripts will easily be able to spot them. For those who are not, the `obj-$(CONFIG_FOO)` entries you see everywhere expand into `obj-y` or `obj-m`, depending on whether the `CONFIG_FOO` variable has been set to `y` or `m`. While we are at it, those were exactly the kind of variables that you have set in the `.config` file in the top-level directory of Linux kernel source tree, the last time when you said `make menuconfig` or something like that.

4.3 The __init and __exit Macros

The `__init` macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers, but not loadable modules. If you think about when the init function is invoked, this makes perfect sense.

There is also an `__initdata` which works similarly to `__init` but for init variables rather than functions.

The `__exit` macro causes the omission of the function when the module is built into the kernel, and like `__init`, has no effect for loadable modules. Again, if you consider when the cleanup function runs, this makes complete sense; built-in drivers do not need a cleanup function, while loadable modules do.

These macros are defined in include/linux/init.h and serve to free up kernel memory. When you boot your kernel and see something like Freeing unused kernel memory: 236k freed, this is precisely what the kernel is freeing.

```c
/*
 * hello-3.c - Illustrating the __init, __initdata and __exit macros.
 */
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
    pr_info("Hello, world %d\n", hello3_data);
    return 0;
}

static void __exit hello_3_exit(void)
{
    pr_info("Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);

MODULE_LICENSE("GPL");
```

4.4 Licensing and Module Documentation

Honestly, who loads or even cares about proprietary modules? If you do then you might have seen something like this:

```
$ sudo insmod xxxxxx.ko
loading out-of-tree module taints kernel.
module license 'unspecified' taints kernel.
```

You can use a few macros to indicate the license for your module. Some examples are "GPL", "GPL v2", "GPL and additional rights", "Dual BSD/GPL", "Dual MIT/GPL", "Dual MPL/GPL" and "Proprietary". They are defined within include/linux/module.h.

To reference what license you're using a macro is available called `MODULE_LICENSE`. This and a few other macros describing the module are illustrated in the below example.

```c
/*
 * hello-4.c - Demonstrates module documentation.
 */
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("LKMPG");
MODULE_DESCRIPTION("A sample driver");

static int __init init_hello_4(void)
{
    pr_info("Hello, world 4\n");
    return 0;
}

static void __exit cleanup_hello_4(void)
{
    pr_info("Goodbye, world 4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);
```

4.5 Passing Command Line Arguments to a Module

Modules can take command line arguments, but not with the argc/argv you might be used to.

To allow arguments to be passed to your module, declare the variables that will take the values of the command line arguments as global and then use the `module_param()` macro, (defined in include/linux/moduleparam.h) to set the mechanism up. At runtime, `insmod` will fill the variables with any command line arguments that are given, like `insmod mymodule.ko myvariable=5`. The variable declarations and macros should be placed at the beginning of the module for clarity. The example code should clear up my admittedly lousy explanation.

The `module_param()` macro takes 3 arguments: the name of the variable, its type and permissions for the corresponding file in sysfs. Integer types can be signed as usual or unsigned. If you'd like to use arrays of integers or strings see `module_param_array()` and `module_param_string()`.

```
int myint = 3;
module_param(myint, int, 0);
```

Arrays are supported too, but things are a bit different now than they were in the olden days. To keep track of the number of parameters you need to pass a pointer to a count variable as third parameter. At your option, you could also ignore the count and pass `NULL` instead. We show both possibilities here:

```
int myintarray[2];
module_param_array(myintarray, int, NULL, 0); /* not interested in count */

short myshortarray[4];
int count;
module_param_array(myshortarray, short, &count, 0); /* put count into "count"
↪  variable */
```

A good use for this is to have the module variable's default values set, like a port or IO address. If the variables contain the default values, then perform autodetection (explained elsewhere). Otherwise, keep the current value. This will be made clear later on.

Lastly, there is a macro function, `MODULE_PARM_DESC()`, that is used to document arguments that the module can take. It takes two parameters: a variable name and a free form string describing that variable.

```
/*
 * hello-5.c - Demonstrates command line argument passing to a module.
 */
#include <linux/init.h>
#include <linux/kernel.h> /* for ARRAY_SIZE() */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/printk.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";
static int myintarray[2] = { 420, 420 };
static int arr_argc = 0;
```

```c

/* module_param(foo, int, 0000)
 * The first param is the parameter's name.
 * The second param is its data type.
 * The final argument is the permissions bits,
 * for exposing parameters in sysfs (if non-zero) at a later stage.
 */
module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

/* module_param_array(name, type, num, perm);
 * The first param is the parameter's (in this case the array's) name.
 * The second param is the data type of the elements of the array.
 * The third argument is a pointer to the variable that will store the number
 * of elements of the array initialized by the user at module loading time.
 * The fourth argument is the permission bits.
 */
module_param_array(myintarray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintarray, "An array of integers");

static int __init hello_5_init(void)
{
    int i;

    pr_info("Hello, world 5\n=============\n");
    pr_info("myshort is a short integer: %hd\n", myshort);
    pr_info("myint is an integer: %d\n", myint);
    pr_info("mylong is a long integer: %ld\n", mylong);
    pr_info("mystring is a string: %s\n", mystring);

    for (i = 0; i < ARRAY_SIZE(myintarray); i++)
        pr_info("myintarray[%d] = %d\n", i, myintarray[i]);

    pr_info("got %d arguments for myintarray.\n", arr_argc);
    return 0;
}

static void __exit hello_5_exit(void)
{
    pr_info("Goodbye, world 5\n");
}

module_init(hello_5_init);
module_exit(hello_5_exit);
```

It is recommended to experiment with the following code:

```
$ sudo insmod hello-5.ko mystring="bebop" myintarray=-1
$ sudo dmesg -t | tail -7
myshort is a short integer: 1
```

```
myint is an integer: 420
mylong is a long integer: 9999
mystring is a string: bebop
myintarray[0] = -1
myintarray[1] = 420
got 1 arguments for myintarray.

$ sudo rmmod hello-5
$ sudo dmesg -t | tail -1
Goodbye, world 5

$ sudo insmod hello-5.ko mystring="supercalifragilisticexpialidocious" myintarray=-1,-1
$ sudo dmesg -t | tail -7
myshort is a short integer: 1
myint is an integer: 420
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious
myintarray[0] = -1
myintarray[1] = -1
got 2 arguments for myintarray.

$ sudo rmmod hello-5
$ sudo dmesg -t | tail -1
Goodbye, world 5

$ sudo insmod hello-5.ko mylong=hello
insmod: ERROR: could not insert module hello-5.ko: Invalid parameters
```

4.6 Modules Spanning Multiple Files

Sometimes it makes sense to divide a kernel module between several source files.

Here is an example of such a kernel module.

```c
/*
 * start.c - Illustration of multi filed modules
 */

#include <linux/kernel.h> /* We are doing kernel work */
#include <linux/module.h> /* Specifically, a module */

int init_module(void)
{
    pr_info("Hello, world - this is the kernel speaking\n");
    return 0;
}

MODULE_LICENSE("GPL");
```

The next file:

```c
/*
 * stop.c - Illustration of multi filed modules
 */

#include <linux/kernel.h> /* We are doing kernel work */
#include <linux/module.h> /* Specifically, a module  */

void cleanup_module(void)
{
    pr_info("Short is the life of a kernel module\n");
}

MODULE_LICENSE("GPL");
```

And finally, the makefile:

```makefile
obj-m += hello-1.o
obj-m += hello-2.o
obj-m += hello-3.o
obj-m += hello-4.o
obj-m += hello-5.o
obj-m += startstop.o
startstop-objs := start.o stop.o

PWD := $(CURDIR)

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

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

This is the complete makefile for all the examples we have seen so far. The first five lines are nothing special, but for the last example we will need two lines. First we invent an object name for our combined module, second we tell `make` what object files are part of that module.

4.7 Building modules for a precompiled kernel

Obviously, we strongly suggest you to recompile your kernel, so that you can enable a number of useful debugging features, such as forced module unloading (`MODULE_FORCE_UNLOAD`): when this option is enabled, you can force the kernel to unload a module even when it believes it is unsafe, via a `sudo rmmod -f module` command. This option can save you a lot of time and a number of reboots during the development of a module. If you do not want to recompile your kernel then you should consider running the examples within a test distribution on a virtual machine. If you mess anything up then you can easily reboot or restore the virtual machine (VM).

There are a number of cases in which you may want to load your module into a precompiled running kernel, such as the ones shipped with common Linux distributions, or a kernel you have compiled in the past. In certain circumstances you could require to compile and insert a module into a running kernel which you are not allowed to recompile, or on a machine that you prefer not to reboot. If you can't think of a case that will force you to use modules for a precompiled kernel you might want to skip this and treat the rest of this chapter as a big footnote.

Now, if you just install a kernel source tree, use it to compile your kernel module and you try to insert your module into the kernel, in most cases you would obtain an error as follows:

```
insmod: ERROR: could not insert module poet.ko: Invalid module format
```

Less cryptic information is logged to the systemd journal:

```
kernel: poet: disagrees about version of symbol module_layout
```

In other words, your kernel refuses to accept your module because version strings (more precisely, *version magic*, see include/linux/vermagic.h) do not match. Incidentally, version magic strings are stored in the module object in the form of a static string, starting with `vermagic:`. Version data are inserted in your module when it is linked against the `kernel/module.o` file. To inspect version magics and other strings stored in a given module, issue the command `modinfo module.ko`:

```
$ modinfo hello-4.ko
description:    A sample driver
author:         LKMPG
license:        GPL
srcversion:     B2AA7FBFCC2C39AED665382
depends:
retpoline:      Y
name:           hello_4
vermagic:       5.4.0-70-generic SMP mod_unload modversions
```

To overcome this problem we could resort to the `--force-vermagic` option, but this solution is potentially unsafe, and unquestionably unacceptable in production modules. Consequently, we want to compile our module in an environment which was identical to the one in which our precompiled kernel was built. How to do this, is the subject of the remainder of this chapter.

First of all, make sure that a kernel source tree is available, having exactly the same version as your current kernel. Then, find the configuration file which was used to compile your precompiled kernel. Usually, this is available in your current `boot` directory, under a name like `config-5.14.x`. You may just want to copy it to your kernel source tree: `` cp /boot/config-`uname -r` .config ``.

Let's focus again on the previous error message: a closer look at the version magic strings suggests that, even with two configuration files which are exactly the same, a slight difference in the version magic could be possible, and it is sufficient to prevent insertion of the module into the kernel. That slight difference, namely the custom string which appears in the module's version magic and not in the kernel's one, is due to a modification with respect to the original, in the makefile that some distributions include. Then, examine your `Makefile`, and make sure that the specified version information matches exactly the one used for your current kernel. For example, your makefile could start as follows:

```
VERSION = 5
PATCHLEVEL = 14
SUBLEVEL = 0
EXTRAVERSION = -rc2
```

In this case, you need to restore the value of symbol **EXTRAVERSION** to **-rc2**. We suggest keeping a backup copy of the makefile used to compile your kernel available in `/lib/modules/5.14.0-rc2/build`. A simple command as following should suffice.

```
cp /lib/modules/`uname -r`/build/Makefile linux-`uname -r`
```

Here ``linux-`uname -r` `` is the Linux kernel source you are attempting to build.

Now, please run `make` to update configuration and version headers and objects:

```
$ make
  SYNC    include/config/auto.conf.cmd
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/confdata.o
  HOSTCC  scripts/kconfig/expr.o
  LEX     scripts/kconfig/lexer.lex.c
  YACC    scripts/kconfig/parser.tab.[ch]
  HOSTCC  scripts/kconfig/preprocess.o
  HOSTCC  scripts/kconfig/symbol.o
  HOSTCC  scripts/kconfig/util.o
  HOSTCC  scripts/kconfig/lexer.lex.o
  HOSTCC  scripts/kconfig/parser.tab.o
  HOSTLD  scripts/kconfig/conf
```

If you do not desire to actually compile the kernel, you can interrupt the build process (CTRL-C) just after the SPLIT line, because at that time, the files you need are ready. Now you can turn back to the directory of your module and compile it: It will be built exactly according to your current kernel settings, and it will load into it without any errors.

5 Preliminaries

5.1 How modules begin and end

A typical program starts with a `main()` function, executes a series of instructions, and terminates after completing these instructions. Kernel modules, however, follow a different pattern. A module always begins with either the `init_module` function or a function designated by the `module_init` call. This function acts as the module's entry point, informing the kernel of the module's functionalities and preparing the kernel to utilize the module's functions when necessary. After performing these tasks, the entry function returns, and the module remains inactive until the kernel requires its code.

All modules conclude by invoking either `cleanup_module` or a function specified through the `module_exit` call. This serves as the module's exit function, reversing the actions of the entry function by unregistering the previously registered functionalities.

It is mandatory for every module to have both an entry and an exit function. While there are multiple methods to define these functions, the terms "entry function" and "exit function" are generally used. However, they may occasionally be referred to as `init_module` and `cleanup_module`, which are understood to mean the same.

5.2 Functions available to modules

Programmers use functions they do not define all the time. A prime example of this is `printf()`. You use these library functions which are provided by the standard C library, libc. The definitions for these functions do not actually enter your program until the linking stage, which ensures that the code (for `printf()` for example) is available, and fixes the call instruction to point to that code.

Kernel modules are different here, too. In the hello world example, you might have noticed that we used a function, `pr_info()` but did not include a standard I/O library. That is because modules are object files whose symbols get resolved upon running `insmod` or `modprobe`. The definition for the symbols comes from the kernel itself; the only external functions you can use are the ones provided by the kernel. If you're curious about what symbols have been exported by your kernel, take a look at `/proc/kallsyms`.

One point to keep in mind is the difference between library functions and system calls. Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions that do the real work — system calls. System calls run in kernel mode on the user's behalf and are provided by the kernel itself. The library function `printf()` may look like a very general printing function, but all it really does is format the data into strings and write the string data using the low-level system call `write()`, which then sends the data to standard output.

Would you like to see what system calls are made by `printf()`? It is easy! Compile the following program:

```c
#include <stdio.h>

int main(void)
{
    printf("hello");
    return 0;
}
```

with `gcc -Wall -o hello hello.c`. Run the executable with `strace ./hello`. Are you impressed? Every line you see corresponds to a system call. strace is a handy program that gives you details about what system calls a program is making, including which call is made, what its arguments are and what it returns. It is an invaluable tool for figuring out things like what files a program is trying to access. Towards the end, you will see a line which looks like `write(1, "hello", 5hello)`. There it is. The face behind the `printf()` mask. You may not be familiar with write, since most people use library functions for file I/O (like `fopen`, `fputs`, `fclose`). If that is the case, try looking at man 2 write. The 2nd man section is devoted to system calls (like `kill()` and `read()`). The 3rd man section is devoted to library calls, which you would probably be more familiar with (like `cosh()` and `random()`).

You can even write modules to replace the kernel's system calls, which we will do shortly. Crackers often make use of this sort of thing for backdoors or trojans, but you can write your own modules to do more benign things, like have the kernel write Tee hee, that tickles! every time someone tries to delete a file on your system.

5.3 User Space vs Kernel Space

The kernel primarily manages access to resources, be it a video card, hard drive, or memory. Programs frequently vie for the same resources. For instance, as a document is saved, updatedb might commence updating the locate database. Sessions in editors like vim and processes like updatedb can simultaneously utilize the hard drive. The kernel's role is to maintain order, ensuring that users do not access resources indiscriminately.

To manage this, CPUs operate in different modes, each offering varying levels of system control. The Intel 80386 architecture, for example, featured four such modes, known as rings. Unix, however, utilizes only two of these rings: the highest ring (ring 0, also known as "supervisor mode", where all actions are permissible) and the lowest ring, referred to as "user mode".

Recall the discussion about library functions vs system calls. Typically, you use a library function in user mode. The library function calls one or more system calls, and these system calls execute on the library function's behalf, but do so in supervisor mode since they are part of the kernel itself. Once the system call completes its task, it returns and execution gets transferred back to user mode.

5.4 Name Space

When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you are writing routines which will be part of a bigger problem, any global variables you have are part of a community of other peoples' global variables; some of the variable names can clash. When a program has lots of global variables which aren't meaningful enough to be distinguished, you get namespace pollution. In large projects, effort must be made to remember reserved names, and to find ways to develop a scheme for naming unique variable names and symbols.

When writing kernel code, even the smallest module will be linked against the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well-defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you do not want to declare everything as static, another option is to declare a symbol table and register it with the kernel. We will get to this later.

The file `/proc/kallsyms` holds all the symbols that the kernel knows about and which are therefore accessible to your modules since they share the kernel's codespace.

5.5 Code space

Memory management is a very complicated subject and the majority of O'Reilly's Understanding The Linux Kernel exclusively covers memory management! We are not setting out to be experts on memory managements, but we do need to know a couple of facts to even begin worrying about writing real modules.

If you have not thought about what a segfault really means, you may be surprised to hear that pointers do not actually point to memory locations. Not real ones, anyway. When a process is created, the kernel sets aside a portion of real physical memory and hands it to the process to use for its executing code, variables, stack, heap and other things which a computer scientist would know about. This memory begins with 0x00000000 and extends up to whatever it needs to be. Since the memory space for any two processes do not overlap, every process that can access a memory address, say 0xbffff978, would be accessing a different location in real physical memory! The processes would be accessing an index named 0xbffff978 which points to some kind of offset into the region of memory set aside for that particular process. For the most part, a process like our Hello, World program can't access the space of another process, although there are ways which we will talk about later.

The kernel has its own space of memory as well. Since a module is code which can be dynamically inserted and removed in the kernel (as opposed to a semi-autonomous object), it shares the kernel's codespace rather than having its own. Therefore, if your module segfaults, the kernel segfaults. And if you start writing over data because of an off-by-one error, then you're trampling on kernel data (or code). This is even worse than it sounds, so try your best to be careful.

It should be noted that the aforementioned discussion applies to any operating system utilizing a monolithic kernel. This concept differs slightly from *"building all your modules into the kernel"*, although the underlying principle is similar. In contrast, there are microkernels, where modules are allocated their own code space. Two notable examples of microkernels include the GNU Hurd and the Zircon kernel of Google's Fuchsia.

5.6 Device Drivers

One class of module is the device driver, which provides functionality for hardware like a serial port. On Unix, each piece of hardware is represented by a file located in `/dev` named a device file which provides the means to communicate with the hardware. The device driver provides the communication on behalf of a user program. So the es1370.ko sound card device driver might connect the `/dev/sound` device file to the Ensoniq IS1370 sound card. A userspace program like mp3blaster can use `/dev/sound` without ever knowing what kind of sound card is installed.

Let's look at some device files. Here are device files which represent the first three partitions on the primary master IDE hard drive:

```
$ ls -l /dev/hda[1-3]
brw-rw----  1 root  disk  3, 1 Jul  5  2000 /dev/hda1
brw-rw----  1 root  disk  3, 2 Jul  5  2000 /dev/hda2
brw-rw----  1 root  disk  3, 3 Jul  5  2000 /dev/hda3
```

Notice the column of numbers separated by a comma. The first number is called the device's major number. The second number is the minor number. The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. All the above major numbers are 3, because they're all controlled by the same driver.

The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all three devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.

Devices are divided into two types: character devices and block devices. The difference is that block devices have a buffer for requests, so they can choose the best order in which to respond to the requests. This is important in the case of storage devices, where it is faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of `ls -l`. If it is 'b' then it is a block device, and if it is 'c' then it is a character device. The devices you see above are block devices. Here are some character devices (the serial ports):

```
crw-rw----  1 root  dial 4, 64 Feb 18 23:34 /dev/ttyS0
crw-r-----  1 root  dial 4, 65 Nov 17 10:26 /dev/ttyS1
crw-rw----  1 root  dial 4, 66 Jul  5  2000 /dev/ttyS2
crw-rw----  1 root  dial 4, 67 Jul  5  2000 /dev/ttyS3
```

If you want to see which major numbers have been assigned, you can look at Documentation/admin-guide/devices.txt.

When the system was installed, all of those device files were created by the `mknod` command. To create a new char device named `coffee` with major/minor number 12 and 2, simply do `mknod /dev/coffee c 12 2`. You do not have to put your device files into `/dev`, but it is done by convention. Linus put his device files in `/dev`, and so should you. However, when creating a device file for testing purposes, it is probably OK to place it in your working directory where you compile the kernel module. Just be sure to put it in the right place when you're done writing the device driver.

A few final points, although implicit in the previous discussion, are worth stating explicitly for clarity. When a device file is accessed, the kernel utilizes the file's major number to identify the appropriate driver for handling the access. This indicates that the kernel does not necessarily rely on or need to be aware of the minor number. It is the driver that concerns itself with the minor number, using it to differentiate between various pieces of hardware.

It is important to note that when referring to *"hardware"*, the term is used in a slightly more abstract sense than just a physical PCI card that can be held in hand. Consider the following two device files:

```
$ ls -l /dev/sda /dev/sdb
brw-rw---- 1 root disk 8,  0 Jan  3 09:02 /dev/sda
brw-rw---- 1 root disk 8, 16 Jan  3 09:02 /dev/sdb
```

By now you can look at these two device files and know instantly that they are block devices and are handled by same driver (block major 8). Sometimes two device files with the same major but different minor number can actually represent the same piece of physical hardware. So just be aware that the word "hardware" in our discussion can mean something very abstract.

6 Character Device drivers

6.1 The file_operations Structure

The `file_operations` structure is defined in include/linux/fs.h, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.

For example, every character driver needs to define a function that reads from the device. The `file_operations` structure holds the address of the module's function that performs that operation. Here is what the definition looks like for kernel 5.4:

```c
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
        loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
        loff_t, size_t, unsigned int);
    loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
             struct file *file_out, loff_t pos_out,
             loff_t len, unsigned int remap_flags);
    int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

Some operations are not implemented by a driver. For example, a driver that handles a video card will not need to read from a directory structure. The corresponding entries in the `file_operations` structure should be set to `NULL`.

There is a gcc extension that makes assigning to this structure more convenient. You will see it in modern drivers, and may catch you by surprise. This is what the new way of assigning to the structure looks like:

```c
struct file_operations fops = {
	read: device_read,
	write: device_write,
	open: device_open,
	release: device_release
};
```

However, there is also a C99 way of assigning to elements of a structure, designated initializers, and this is definitely preferred over using the GNU extension. You should use this syntax in case someone wants to port your driver. It will help with compatibility:

```c
struct file_operations fops = {
	.read = device_read,
	.write = device_write,
	.open = device_open,
	.release = device_release
};
```

The meaning is clear, and you should be aware that any member of the structure which you do not explicitly assign will be initialized to `NULL` by gcc.

An instance of `struct file_operations` containing pointers to functions that are used to implement `read`, `write`, `open`, ... system calls is commonly named `fops`.

Since Linux v3.14, the read, write and seek operations are guaranteed for thread-safe by using the `f_pos` specific lock, which makes the file position update to become the mutual exclusion. So, we can safely implement those operations without unnecessary locking.

Additionally, since Linux v5.6, the `proc_ops` structure was introduced to replace the use of the `file_operations` structure when registering proc handlers. See more information in the 7.1 section.

6.2 The file structure

Each device is represented in the kernel by a file structure, which is defined in include/linux/fs.h. Be aware that a file is a kernel level structure and never appears in a user space program. It is not the same thing as a `FILE`, which is defined by glibc and would never appear in a kernel space function. Also, its name is a bit misleading; it represents an abstract open 'file', not a file on a disk, which is represented by a structure named `inode`.

An instance of struct file is commonly named `filp`. You'll also see it referred to as a struct file object. Resist the temptation.

Go ahead and look at the definition of file. Most of the entries you see, like struct dentry are not used by device drivers, and you can ignore them. This is because drivers do not fill file directly; they only use structures contained in file which are created elsewhere.

6.3 Registering A Device

As discussed earlier, char devices are accessed through device files, usually located in `/dev`. This is by convention. When writing a driver, it is OK to put the device file in your current directory. Just make sure you place it in `/dev` for a production driver. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it is operating on, just in case the driver handles more than one device.

Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization. You do this by using the `register_chrdev` function, defined by include/linux/fs.h.

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

Where unsigned int major is the major number you want to request, `const char *name` is the name of the device as it will appear in `/proc/devices` and `struct file_operations *fops` is a pointer to the `file_operations` table for your driver. A negative return value means the registration failed. Note that we didn't pass the minor number to `register_chrdev`. That is because the kernel doesn't care about the minor number; only our driver uses it.

Now the question is, how do you get a major number without hijacking one that's already in use? The easiest way would be to look through Documentation/admin-guide/devices.txt and pick an unused one. That is a bad way of doing things because you will never be sure if the number you picked will be assigned later. The answer is that you can ask the kernel to assign you a dynamic major number.

If you pass a major number of 0 to `register_chrdev`, the return value will be the dynamically allocated major number. The downside is that you can not make a device file in advance, since you do not know what the major number will be. There are a couple of ways to do this. First, the driver itself can print the newly assigned number and we can make the device file by hand. Second, the newly registered device will have an entry in `/proc/devices`, and we can either make the device file by hand or write a shell script to read the file in and make the device file. The third method is that we can have our driver make the device file using the `device_create` function after a successful registration and `device_destroy` during the call to `cleanup_module`.

However, `register_chrdev()` would occupy a range of minor numbers associated with the given major. The recommended way to reduce waste for char device registration is using cdev interface.

The newer interface completes the char device registration in two distinct steps. First, we should register a range of device numbers, which can be completed with `register_chrdev_region` or `alloc_chrdev_region`.

```c
int register_chrdev_region(dev_t from, unsigned count, const char *name);
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const
↪  char *name);
```

The choice between two different functions depends on whether you know the major numbers for your device. Using `register_chrdev_region` if you know the device major number and `alloc_chrdev_region` if you would like to allocate a dynamically-allocated major number.

Second, we should initialize the data structure `struct cdev` for our char device and associate it with the device numbers. To initialize the `struct cdev`, we can achieve by the similar sequence of the following codes.

```c
struct cdev *my_dev = cdev_alloc();
my_cdev->ops = &my_fops;
```

However, the common usage pattern will embed the `struct cdev` within a device-specific structure of your own. In this case, we'll need `cdev_init` for the initialization.

```c
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

Once we finish the initialization, we can add the char device to the system by using the `cdev_add`.

```c
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

To find an example using the interface, you can see `ioctl.c` described in section 9.

6.4 Unregistering A Device

We can not allow the kernel module to be `rmmod`'ed whenever root feels like it. If the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be. If we are lucky, no other code was loaded there, and we'll get an ugly error message. If we are unlucky, another kernel module was loaded into the same location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can not be very positive.

Normally, when you do not want to allow something, you return an error code (a negative number) from the function which is supposed to do it. With `cleanup_module` that's impossible because it is a void function. However, there is a counter which keeps track of how many processes are using your module.

You can see what its value is by looking at the 3rd field with the command `cat /proc/modules` or `sudo lsmod`. If this number isn't zero, `rmmod` will fail. Note that you do not have to check the counter within `cleanup_module` because the check will be performed for you by the system call `sys_delete_module`, defined in include/linux/syscalls.h. You should not use this counter directly, but there are functions defined in include/linux/module.h which let you increase, decrease and display this counter:

- `try_module_get(THIS_MODULE)`: Increment the reference count of current module.
- `module_put(THIS_MODULE)`: Decrement the reference count of current module.
- `module_refcount(THIS_MODULE)`: Return the value of reference count of current module.

It is important to keep the counter accurate; if you ever do lose track of the correct usage count, you will never be able to unload the module; it's now reboot time, boys and girls. This is bound to happen to you sooner or later during a module's development.

6.5 chardev.c

The next code sample creates a char driver named `chardev`. You can dump its device file.

```
cat /proc/devices
```

(or open the file with a program) and the driver will put the number of times the device file has been read from into the file. We do not support writing to the file (like `echo "hi" > /dev/hello`), but catch these attempts and tell the user that the operation is not supported. Don't worry if you don't see what we do with the data we read into the buffer; we don't do much with it. We simply read in the data and print a message acknowledging that we received it.

In the multiple-threaded environment, without any protection, concurrent access to the same memory may lead to the race condition, and will not preserve the performance. In the kernel module, this problem may happen due to multiple instances accessing the shared resources. Therefore, a solution is to enforce the exclusive access. We use atomic Compare-And-Swap (CAS) to maintain the states, `CDEV_NOT_USED` and `CDEV_EXCLUSIVE_OPEN`, to determine whether the file is currently opened by someone or not. CAS compares the contents of a memory location with the expected value and, only if they are the same, modifies the contents of that memory location to the desired value. See more concurrency details in the 12 section.

```c
/*
 * chardev.c: Creates a read-only char device that says how many times
 * you have read from the dev file
 */

#include <linux/atomic.h>
#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kernel.h> /* for sprintf() */
#include <linux/module.h>
#include <linux/printk.h>
#include <linux/types.h>
#include <linux/uaccess.h> /* for get_user and put_user */
#include <linux/version.h>

#include <asm/errno.h>

/*  Prototypes - this would normally go in a .h file */
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char __user *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char __user *, size_t,
                            loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80 /* Max length of the message from the device */

/* Global variables are declared as static, so are global within the file. */

static int major; /* major number assigned to our device driver */

enum {
    CDEV_NOT_USED = 0,
    CDEV_EXCLUSIVE_OPEN = 1,
};

/* Is device open? Used to prevent multiple access to device */
static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);

static char msg[BUF_LEN + 1]; /* The msg the device will give when asked */

static struct class *cls;

static struct file_operations chardev_fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release,
};

static int __init chardev_init(void)
{
```

```
    major = register_chrdev(0, DEVICE_NAME, &chardev_fops);

    if (major < 0) {
        pr_alert("Registering char device failed with %d\n", major);
        return major;
    }

    pr_info("I was assigned major number %d.\n", major);

#if LINUX_VERSION_CODE >= KERNEL_VERSION(6, 4, 0)
    cls = class_create(DEVICE_NAME);
#else
    cls = class_create(THIS_MODULE, DEVICE_NAME);
#endif
    device_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_NAME);

    return SUCCESS;
}

static void __exit chardev_exit(void)
{
    device_destroy(cls, MKDEV(major, 0));
    class_destroy(cls);

    /* Unregister the device */
    unregister_chrdev(major, DEVICE_NAME);
}

/* Methods */

/* Called when a process tries to open the device file, like
 * "sudo cat /dev/chardev"
 */
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;

    if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
        return -EBUSY;

    sprintf(msg, "I already told you %d times Hello world!\n", counter++);
    try_module_get(THIS_MODULE);

    return SUCCESS;
}

/* Called when a process closes the device file. */
static int device_release(struct inode *inode, struct file *file)
{
    /* We're now ready for our next caller */
    atomic_set(&already_open, CDEV_NOT_USED);

    /* Decrement the usage count, or else once you opened the file, you will
     * never get rid of the module.
     */
```

```c
    module_put(THIS_MODULE);

    return SUCCESS;
}

/* Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp, /* see include/linux/fs.h   */
                           char __user *buffer, /* buffer to fill with data */
                           size_t length, /* length of the buffer     */
                           loff_t *offset)
{
    /* Number of bytes actually written to the buffer */
    int bytes_read = 0;
    const char *msg_ptr = msg;

    if (!*(msg_ptr + *offset)) { /* we are at the end of message */
        *offset = 0; /* reset the offset */
        return 0; /* signify end of file */
    }

    msg_ptr += *offset;

    /* Actually put the data into the buffer */
    while (length && *msg_ptr) {
        /* The buffer is in the user data segment, not the kernel
         * segment so "*" assignment won't work.  We have to use
         * put_user which copies data from the kernel data segment to
         * the user data segment.
         */
        put_user(*(msg_ptr++), buffer++);
        length--;
        bytes_read++;
    }

    *offset += bytes_read;

    /* Most read functions return the number of bytes put into the buffer. */
    return bytes_read;
}

/* Called when a process writes to dev file: echo "hi" > /dev/hello */
static ssize_t device_write(struct file *filp, const char __user *buff,
                            size_t len, loff_t *off)
{
    pr_alert("Sorry, this operation is not supported.\n");
    return -EINVAL;
}

module_init(chardev_init);
module_exit(chardev_exit);

MODULE_LICENSE("GPL");
```

6.6 Writing Modules for Multiple Kernel Versions

The system calls, which are the major interface the kernel shows to the processes, generally stay the same across versions. A new system call may be added, but usually the old ones will behave exactly like they used to. This is necessary for backward compatibility – a new kernel version is not supposed to break regular processes. In most cases, the device files will also remain the same. On the other hand, the internal interfaces within the kernel can and do change between versions.

There are differences between different kernel versions, and if you want to support multiple kernel versions, you will find yourself having to code conditional compilation directives. The way to do this to compare the macro `LINUX_VERSION_CODE` to the macro `KERNEL_VERSION`. In version `a.b.c` of the kernel, the value of this macro would be $2^{16}a + 2^{8}b + c$.

7 The /proc File System

In Linux, there is an additional mechanism for the kernel and kernel modules to send information to processes — the `/proc` file system. Originally designed to allow easy access to information about processes (hence the name), it is now used by every bit of the kernel which has something interesting to report, such as `/proc/modules` which provides the list of modules and `/proc/meminfo` which gathers memory usage statistics.

The method to use the proc file system is very similar to the one used with device drivers — a structure is created with all the information needed for the `/proc` file, including pointers to any handler functions (in our case there is only one, the one called when somebody attempts to read from the `/proc` file). Then, `init_module` registers the structure with the kernel and `cleanup_module` unregisters it.

Normal file systems are located on a disk, rather than just in memory (which is where `/proc` is), and in that case the index-node (inode for short) number is a pointer to a disk location where the file's inode is located. The inode contains information about the file, for example the file's permissions, together with a pointer to the disk location or locations where the file's data can be found.

Because we don't get called when the file is opened or closed, there's nowhere for us to put `try_module_get` and `module_put` in this module, and if the file is opened and then the module is removed, there's no way to avoid the consequences.

Here a simple example showing how to use a `/proc` file. This is the HelloWorld for the `/proc` filesystem. There are three parts: create the file `/proc/helloworld` in the function `init_module`, return a value (and a buffer) when the file `/proc/helloworld` is read in the callback function `procfile_read`, and delete the file `/proc/helloworld` in the function `cleanup_module`.

The `/proc/helloworld` is created when the module is loaded with the function `proc_create`. The return value is a pointer to `struct proc_dir_entry`, and it will be used to configure the file /proc/helloworld (for example, the owner of this file). A null return value means that the creation has failed.

Every time the file /proc/helloworld is read, the function procfile_read is called. Two parameters of this function are very important: the buffer (the second parameter) and the offset (the fourth one). The content of the buffer will be returned to the application which read it (for example the cat command). The offset is the current position in the file. If the return value of the function is not null, then this function is called again. So be careful with this function, if it never returns zero, the read function is called endlessly.

```
$ cat /proc/helloworld
HelloWorld!
```

```c
/*
 * procfs1.c
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>
#include <linux/version.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

#define procfs_name "helloworld"

static struct proc_dir_entry *our_proc_file;

static ssize_t procfile_read(struct file *file_pointer, char __user *buffer,
                             size_t buffer_length, loff_t *offset)
{
    char s[13] = "HelloWorld!\n";
    int len = sizeof(s);
    ssize_t ret = len;

    if (*offset >= len || copy_to_user(buffer, s, len)) {
        pr_info("copy_to_user failed\n");
        ret = 0;
    } else {
        pr_info("procfile read %s\n",
        ↪  file_pointer->f_path.dentry->d_name.name);
        *offset += len;
    }

    return ret;
}

#ifdef HAVE_PROC_OPS
static const struct proc_ops proc_file_fops = {
    .proc_read = procfile_read,
};
#else
```

```
static const struct file_operations proc_file_fops = {
    .read = procfile_read,
};
#endif

static int __init procfs1_init(void)
{
    our_proc_file = proc_create(procfs_name, 0644, NULL, &proc_file_fops);
    if (NULL == our_proc_file) {
        proc_remove(our_proc_file);
        pr_alert("Error:Could not initialize /proc/%s\n", procfs_name);
        return -ENOMEM;
    }

    pr_info("/proc/%s created\n", procfs_name);
    return 0;
}

static void __exit procfs1_exit(void)
{
    proc_remove(our_proc_file);
    pr_info("/proc/%s removed\n", procfs_name);
}

module_init(procfs1_init);
module_exit(procfs1_exit);

MODULE_LICENSE("GPL");
```

7.1 The proc_ops Structure

The `proc_ops` structure is defined in include/linux/proc_fs.h in Linux v5.6+. In older kernels, it used `file_operations` for custom hooks in `/proc` file system, but it contains some members that are unnecessary in VFS, and every time VFS expands `file_operations` set, `/proc` code comes bloated. On the other hand, not only the space, but also some operations were saved by this structure to improve its performance. For example, the file which never disappears in `/proc` can set the `proc_flag` as `PROC_ENTRY_PERMANENT` to save 2 atomic ops, 1 allocation, 1 free in per open/read/close sequence.

7.2 Read and Write a /proc File

We have seen a very simple example for a `/proc` file where we only read the file `/proc/helloworld`. It is also possible to write in a `/proc` file. It works the same way as read, a function is called when the `/proc` file is written. But there is a little difference with read, data comes from user, so you have to import data from user space to kernel space (with `copy_from_user` or `get_user`)

The reason for `copy_from_user` or `get_user` is that Linux memory (on Intel architecture, it may be different under some other processors) is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one for each of the processes.

The only memory segment accessible to a process is its own, so when writing regular programs to run as processes, there is no need to worry about segments. When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system. However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment. The `put_user` and `get_user` macros allow you to access that memory. These functions handle only one character, you can handle several characters with `copy_to_user` and `copy_from_user`. As the buffer (in read or write function) is in kernel space, for write function you need to import data because it comes from user space, but not for the read function because data is already in kernel space.

```
/*
 * procfs2.c -  create a "file" in /proc
 */

#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h> /* Necessary because we use the proc fs */
#include <linux/uaccess.h> /* for copy_from_user */
#include <linux/version.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

#define PROCFS_MAX_SIZE 1024
#define PROCFS_NAME "buffer1k"

/* This structure hold information about the /proc file */
static struct proc_dir_entry *our_proc_file;

/* The buffer used to store character for this module */
static char procfs_buffer[PROCFS_MAX_SIZE];

/* The size of the buffer */
static unsigned long procfs_buffer_size = 0;

/* This function is called then the /proc file is read */
static ssize_t procfile_read(struct file *file_pointer, char __user *buffer,
                             size_t buffer_length, loff_t *offset)
{
    char s[13] = "HelloWorld!\n";
    int len = sizeof(s);
    ssize_t ret = len;

    if (*offset >= len || copy_to_user(buffer, s, len)) {
        pr_info("copy_to_user failed\n");
        ret = 0;
    } else {
```

```
        pr_info("procfile read %s\n",
        ↪  file_pointer->f_path.dentry->d_name.name);
        *offset += len;
    }

    return ret;
}

/* This function is called with the /proc file is written. */
static ssize_t procfile_write(struct file *file, const char __user *buff,
                              size_t len, loff_t *off)
{
    procfs_buffer_size = len;
    if (procfs_buffer_size > PROCFS_MAX_SIZE)
        procfs_buffer_size = PROCFS_MAX_SIZE;

    if (copy_from_user(procfs_buffer, buff, procfs_buffer_size))
        return -EFAULT;

    procfs_buffer[procfs_buffer_size & (PROCFS_MAX_SIZE - 1)] = '\0';
    *off += procfs_buffer_size;
    pr_info("procfile write %s\n", procfs_buffer);

    return procfs_buffer_size;
}

#ifdef HAVE_PROC_OPS
static const struct proc_ops proc_file_fops = {
    .proc_read = procfile_read,
    .proc_write = procfile_write,
};
#else
static const struct file_operations proc_file_fops = {
    .read = procfile_read,
    .write = procfile_write,
};
#endif

static int __init procfs2_init(void)
{
    our_proc_file = proc_create(PROCFS_NAME, 0644, NULL, &proc_file_fops);
    if (NULL == our_proc_file) {
        pr_alert("Error:Could not initialize /proc/%s\n", PROCFS_NAME);
        return -ENOMEM;
    }

    pr_info("/proc/%s created\n", PROCFS_NAME);
    return 0;
}

static void __exit procfs2_exit(void)
{
    proc_remove(our_proc_file);
    pr_info("/proc/%s removed\n", PROCFS_NAME);
}

module_init(procfs2_init);
```

```
module_exit(procfs2_exit);

MODULE_LICENSE("GPL");
```

7.3 Manage /proc file with standard filesystem

We have seen how to read and write a `/proc` file with the `/proc` interface. But it is also possible to manage `/proc` file with inodes. The main concern is to use advanced functions, like permissions.

In Linux, there is a standard mechanism for file system registration. Since every file system has to have its own functions to handle inode and file operations, there is a special structure to hold pointers to all those functions, `struct inode_operations`, which includes a pointer to `struct proc_ops`.

The difference between file and inode operations is that file operations deal with the file itself whereas inode operations deal with ways of referencing the file, such as creating links to it.

In `/proc`, whenever we register a new file, we're allowed to specify which `struct inode_operations` will be used to access to it. This is the mechanism we use, a `struct inode_operations` which includes a pointer to a `struct proc_ops` which includes pointers to our `procfs_read` and `procfs_write` functions.

Another interesting point here is the `module_permission` function. This function is called whenever a process tries to do something with the `/proc` file, and it can decide whether to allow access or not. Right now it is only based on the operation and the uid of the current user (as available in current, a pointer to a structure which includes information on the currently running process), but it could be based on anything we like, such as what other processes are doing with the same file, the time of day, or the last input we received.

It is important to note that the standard roles of read and write are reversed in the kernel. Read functions are used for output, whereas write functions are used for input. The reason for that is that read and write refer to the user's point of view — if a process reads something from the kernel, then the kernel needs to output it, and if a process writes something to the kernel, then the kernel receives it as input.

```
/*
 * procfs3.c
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/uaccess.h>
#include <linux/version.h>
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 10, 0)
#include <linux/minmax.h>
#endif

```

```c
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

#define PROCFS_MAX_SIZE 2048UL
#define PROCFS_ENTRY_FILENAME "buffer2k"

static struct proc_dir_entry *our_proc_file;
static char procfs_buffer[PROCFS_MAX_SIZE];
static unsigned long procfs_buffer_size = 0;

static ssize_t procfs_read(struct file *filp, char __user *buffer,
                           size_t length, loff_t *offset)
{
    if (*offset || procfs_buffer_size == 0) {
        pr_debug("procfs_read: END\n");
        *offset = 0;
        return 0;
    }
    procfs_buffer_size = min(procfs_buffer_size, length);
    if (copy_to_user(buffer, procfs_buffer, procfs_buffer_size))
        return -EFAULT;
    *offset += procfs_buffer_size;

    pr_debug("procfs_read: read %lu bytes\n", procfs_buffer_size);
    return procfs_buffer_size;
}
static ssize_t procfs_write(struct file *file, const char __user *buffer,
                            size_t len, loff_t *off)
{
    procfs_buffer_size = min(PROCFS_MAX_SIZE, len);
    if (copy_from_user(procfs_buffer, buffer, procfs_buffer_size))
        return -EFAULT;
    *off += procfs_buffer_size;

    pr_debug("procfs_write: write %lu bytes\n", procfs_buffer_size);
    return procfs_buffer_size;
}
static int procfs_open(struct inode *inode, struct file *file)
{
    try_module_get(THIS_MODULE);
    return 0;
}
static int procfs_close(struct inode *inode, struct file *file)
{
    module_put(THIS_MODULE);
    return 0;
}

#ifdef HAVE_PROC_OPS
static struct proc_ops file_ops_4_our_proc_file = {
    .proc_read = procfs_read,
    .proc_write = procfs_write,
    .proc_open = procfs_open,
    .proc_release = procfs_close,
};
#else
```

```
static const struct file_operations file_ops_4_our_proc_file = {
    .read = procfs_read,
    .write = procfs_write,
    .open = procfs_open,
    .release = procfs_close,
};
#endif

static int __init procfs3_init(void)
{
    our_proc_file = proc_create(PROCFS_ENTRY_FILENAME, 0644, NULL,
                                &file_ops_4_our_proc_file);
    if (our_proc_file == NULL) {
        pr_debug("Error: Could not initialize /proc/%s\n",
                 PROCFS_ENTRY_FILENAME);
        return -ENOMEM;
    }
    proc_set_size(our_proc_file, 80);
    proc_set_user(our_proc_file, GLOBAL_ROOT_UID, GLOBAL_ROOT_GID);

    pr_debug("/proc/%s created\n", PROCFS_ENTRY_FILENAME);
    return 0;
}

static void __exit procfs3_exit(void)
{
    remove_proc_entry(PROCFS_ENTRY_FILENAME, NULL);
    pr_debug("/proc/%s removed\n", PROCFS_ENTRY_FILENAME);
}

module_init(procfs3_init);
module_exit(procfs3_exit);

MODULE_LICENSE("GPL");
```

Still hungry for procfs examples? Well, first of all keep in mind, there are rumors around, claiming that procfs is on its way out, consider using `sysfs` instead. Consider using this mechanism, in case you want to document something kernel related yourself.

7.4 Manage /proc file with seq_file

As we have seen, writing a `/proc` file may be quite "complex". So to help people writing `/proc` file, there is an API named `seq_file` that helps formatting a `/proc` file for output. It is based on sequence, which is composed of 3 functions: `start()`, `next()`, and `stop()`. The `seq_file` API starts a sequence when a user read the `/proc` file.

A sequence begins with the call of the function `start()`. If the return is a non `NULL` value, the function `next()` is called; otherwise, the `stop()` function is called directly. This function is an iterator, the goal is to go through all the data. Each time `next()` is called, the function `show()` is also called. It writes data values in the buffer read by the user. The function `next()` is called until it returns `NULL`. The sequence ends when `next()` returns `NULL`, then the function `stop()` is called.

BE CAREFUL: when a sequence is finished, another one starts. That means that at the end of function `stop()`, the function `start()` is called again. This loop finishes when the function `start()` returns `NULL`. You can see a scheme of this in the Figure 1.

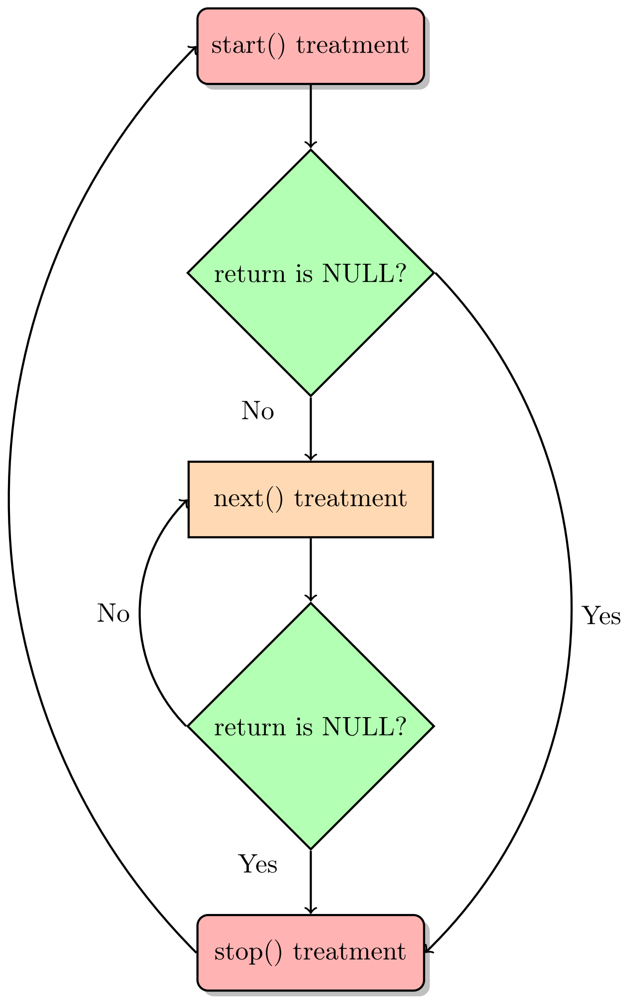


Figure 1: How seq_file works

The `seq_file` provides basic functions for `proc_ops`, such as `seq_read`, `seq_lseek`, and some others. But nothing to write in the `/proc` file. Of course, you can still use the same way as in the previous example.

```
/*
```

```c
 * procfs4.c -  create a "file" in /proc
 * This program uses the seq_file library to manage the /proc file.
 */

#include <linux/kernel.h> /* We are doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h> /* Necessary because we use proc fs */
#include <linux/seq_file.h> /* for seq_file */
#include <linux/version.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

#define PROC_NAME "iter"

/* This function is called at the beginning of a sequence.
 * ie, when:
 *   - the /proc file is read (first time)
 *   - after the function stop (end of sequence)
 */
static void *my_seq_start(struct seq_file *s, loff_t *pos)
{
    static unsigned long counter = 0;

    /* beginning a new sequence? */
    if (*pos == 0) {
        /* yes => return a non null value to begin the sequence */
        return &counter;
    }

    /* no => it is the end of the sequence, return end to stop reading */
    *pos = 0;
    return NULL;
}

/* This function is called after the beginning of a sequence.
 * It is called until the return is NULL (this ends the sequence).
 */
static void *my_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    unsigned long *tmp_v = (unsigned long *)v;
    (*tmp_v)++;
    (*pos)++;
    return NULL;
}

/* This function is called at the end of a sequence. */
static void my_seq_stop(struct seq_file *s, void *v)
{
    /* nothing to do, we use a static value in start() */
}

/* This function is called for each "step" of a sequence. */
static int my_seq_show(struct seq_file *s, void *v)
{
    loff_t *spos = (loff_t *)v;
```

```
    seq_printf(s, "%Ld\n", *spos);
    return 0;
}

/* This structure gather "function" to manage the sequence */
static struct seq_operations my_seq_ops = {
    .start = my_seq_start,
    .next = my_seq_next,
    .stop = my_seq_stop,
    .show = my_seq_show,
};

/* This function is called when the /proc file is open. */
static int my_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &my_seq_ops);
};

/* This structure gather "function" that manage the /proc file */
#ifdef HAVE_PROC_OPS
static const struct proc_ops my_file_ops = {
    .proc_open = my_open,
    .proc_read = seq_read,
    .proc_lseek = seq_lseek,
    .proc_release = seq_release,
};
#else
static const struct file_operations my_file_ops = {
    .open = my_open,
    .read = seq_read,
    .llseek = seq_lseek,
    .release = seq_release,
};
#endif

static int __init procfs4_init(void)
{
    struct proc_dir_entry *entry;

    entry = proc_create(PROC_NAME, 0, NULL, &my_file_ops);
    if (entry == NULL) {
        pr_debug("Error: Could not initialize /proc/%s\n", PROC_NAME);
        return -ENOMEM;
    }

    return 0;
}

static void __exit procfs4_exit(void)
{
    remove_proc_entry(PROC_NAME, NULL);
    pr_debug("/proc/%s removed\n", PROC_NAME);
}

module_init(procfs4_init);
module_exit(procfs4_exit);
```

```
MODULE_LICENSE("GPL");
```

If you want more information, you can read this web page:

- https://lwn.net/Articles/22355/
- https://kernelnewbies.org/Documents/SeqFileHowTo

You can also read the code of fs/seq_file.c in the linux kernel.

8 sysfs: Interacting with your module

sysfs allows you to interact with the running kernel from userspace by reading or setting variables inside of modules. This can be useful for debugging purposes, or just as an interface for applications or scripts. You can find sysfs directories and files under the `/sys` directory on your system.

```
ls -l /sys
```

Attributes can be exported for kobjects in the form of regular files in the filesystem. Sysfs forwards file I/O operations to methods defined for the attributes, providing a means to read and write kernel attributes.

An attribute definition in simply:

```
struct attribute {
    char *name;
    struct module *owner;
    umode_t mode;
};

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

For example, the driver model defines `struct device_attribute` like:

```
struct device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                    const char *buf, size_t count);
};

int device_create_file(struct device *, const struct device_attribute *);
void device_remove_file(struct device *, const struct device_attribute *);
```

To read or write attributes, `show()` or `store()` method must be specified when declaring the attribute. For the common cases include/linux/sysfs.h provides convenience macros (`__ATTR`, `__ATTR_RO`, `__ATTR_WO`, etc.) to make defining attributes easier as well as making code more concise and readable.

An example of a hello world module which includes the creation of a variable accessible via sysfs is given below.

```c
/*
 * hello-sysfs.c sysfs example
 */
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kobject.h>
#include <linux/module.h>
#include <linux/string.h>
#include <linux/sysfs.h>

static struct kobject *mymodule;

/* the variable you want to be able to change */
static int myvariable = 0;

static ssize_t myvariable_show(struct kobject *kobj,
                               struct kobj_attribute *attr, char *buf)
{
    return sprintf(buf, "%d\n", myvariable);
}

static ssize_t myvariable_store(struct kobject *kobj,
                                struct kobj_attribute *attr, char *buf,
                                size_t count)
{
    sscanf(buf, "%du", &myvariable);
    return count;
}

static struct kobj_attribute myvariable_attribute =
    __ATTR(myvariable, 0660, myvariable_show, (void *)myvariable_store);

static int __init mymodule_init(void)
{
    int error = 0;

    pr_info("mymodule: initialized\n");

    mymodule = kobject_create_and_add("mymodule", kernel_kobj);
    if (!mymodule)
        return -ENOMEM;

    error = sysfs_create_file(mymodule, &myvariable_attribute.attr);
    if (error) {
        pr_info("failed to create the myvariable file "
                "in /sys/kernel/mymodule\n");
    }

    return error;
```

```c
}

static void __exit mymodule_exit(void)
{
    pr_info("mymodule: Exit success\n");
    kobject_put(mymodule);
}

module_init(mymodule_init);
module_exit(mymodule_exit);

MODULE_LICENSE("GPL");
```

Make and install the module:

```
make
sudo insmod hello-sysfs.ko
```

Check that it exists:

```
sudo lsmod | grep hello_sysfs
```

What is the current value of `myvariable` ?

```
sudo cat /sys/kernel/mymodule/myvariable
```

Set the value of `myvariable` and check that it changed.

```
echo "32" | sudo tee /sys/kernel/mymodule/myvariable
sudo cat /sys/kernel/mymodule/myvariable
```

Finally, remove the test module:

```
sudo rmmod hello_sysfs
```

In the above case, we use a simple kobject to create a directory under sysfs, and communicate with its attributes. Since Linux v2.6.0, the `kobject` structure made its appearance. It was initially meant as a simple way of unifying kernel code which manages reference counted objects. After a bit of mission creep, it is now the glue that holds much of the device model and its sysfs interface together. For more information about kobject and sysfs, see Documentation/driver-api/driver-model/driver.rst and `https://lwn.net/Articles/51437/`.

9 Talking To Device Files

Device files are supposed to represent physical devices. Most physical devices are used for output as well as input, so there has to be some mechanism for device drivers in the kernel to get the output to send to the device from processes. This is done by opening the device file for output and writing to it, just like writing to a file. In the following example, this is implemented by `device_write`.

This is not always enough. Imagine you had a serial port connected to a modem (even if you have an internal modem, it is still implemented from the CPU's perspective as a serial port connected to a modem, so you don't have to tax your imagination too hard). The natural thing to do would be to use the device file to write things to the modem (either modem commands or data to be sent through the phone line) and read things from the modem (either responses for commands or the data received through the phone line). However, this leaves open the question of what to do when you need to talk to the serial port itself, for example to configure the rate at which data is sent and received.

The answer in Unix is to use a special function called `ioctl` (short for Input Output ConTroL). Every device can have its own `ioctl` commands, which can be read ioctl's (to send information from a process to the kernel), write ioctl's (to return information to a process), both or neither. Notice here the roles of read and write are reversed again, so in ioctl's read is to send information to the kernel and write is to receive information from the kernel.

The ioctl function is called with three parameters: the file descriptor of the appropriate device file, the ioctl number, and a parameter, which is of type long so you can use a cast to use it to pass anything. You will not be able to pass a structure this way, but you will be able to pass a pointer to the structure. Here is an example:

```c
/*
 * ioctl.c
 */
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/ioctl.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/version.h>

struct ioctl_arg {
    unsigned int val;
};

/* Documentation/userspace-api/ioctl/ioctl-number.rst */
#define IOC_MAGIC '\x66'

#define IOCTL_VALSET _IOW(IOC_MAGIC, 0, struct ioctl_arg)
#define IOCTL_VALGET _IOR(IOC_MAGIC, 1, struct ioctl_arg)
#define IOCTL_VALGET_NUM _IOR(IOC_MAGIC, 2, int)
```

```c
#define IOCTL_VALSET_NUM _IOW(IOC_MAGIC, 3, int)

#define IOCTL_VAL_MAXNR 3
#define DRIVER_NAME "ioctltest"

static unsigned int test_ioctl_major = 0;
static unsigned int num_of_dev = 1;
static struct cdev test_ioctl_cdev;
static int ioctl_num = 0;

struct test_ioctl_data {
    unsigned char val;
    rwlock_t lock;
};

static long test_ioctl_ioctl(struct file *filp, unsigned int cmd,
                             unsigned long arg)
{
    struct test_ioctl_data *ioctl_data = filp->private_data;
    int retval = 0;
    unsigned char val;
    struct ioctl_arg data;
    memset(&data, 0, sizeof(data));

    switch (cmd) {
    case IOCTL_VALSET:
        if (copy_from_user(&data, (int __user *)arg, sizeof(data))) {
            retval = -EFAULT;
            goto done;
        }

        pr_alert("IOCTL set val:%x .\n", data.val);
        write_lock(&ioctl_data->lock);
        ioctl_data->val = data.val;
        write_unlock(&ioctl_data->lock);
        break;

    case IOCTL_VALGET:
        read_lock(&ioctl_data->lock);
        val = ioctl_data->val;
        read_unlock(&ioctl_data->lock);
        data.val = val;

        if (copy_to_user((int __user *)arg, &data, sizeof(data))) {
            retval = -EFAULT;
            goto done;
        }

        break;

    case IOCTL_VALGET_NUM:
        retval = __put_user(ioctl_num, (int __user *)arg);
        break;

    case IOCTL_VALSET_NUM:
        ioctl_num = arg;
        break;
```

```c

    default:
        retval = -ENOTTY;
    }

done:
    return retval;
}

static ssize_t test_ioctl_read(struct file *filp, char __user *buf,
                               size_t count, loff_t *f_pos)
{
    struct test_ioctl_data *ioctl_data = filp->private_data;
    unsigned char val;
    int retval;
    int i = 0;

    read_lock(&ioctl_data->lock);
    val = ioctl_data->val;
    read_unlock(&ioctl_data->lock);

    for (; i < count; i++) {
        if (copy_to_user(&buf[i], &val, 1)) {
            retval = -EFAULT;
            goto out;
        }
    }

    retval = count;
out:
    return retval;
}

static int test_ioctl_close(struct inode *inode, struct file *filp)
{
    pr_alert("%s call.\n", __func__);

    if (filp->private_data) {
        kfree(filp->private_data);
        filp->private_data = NULL;
    }

    return 0;
}

static int test_ioctl_open(struct inode *inode, struct file *filp)
{
    struct test_ioctl_data *ioctl_data;

    pr_alert("%s call.\n", __func__);
    ioctl_data = kmalloc(sizeof(struct test_ioctl_data), GFP_KERNEL);

    if (ioctl_data == NULL)
        return -ENOMEM;

    rwlock_init(&ioctl_data->lock);
    ioctl_data->val = 0xFF;
```

```c
    filp->private_data = ioctl_data;

    return 0;
}

static struct file_operations fops = {
#if LINUX_VERSION_CODE < KERNEL_VERSION(6, 4, 0)
    .owner = THIS_MODULE,
#endif
    .open = test_ioctl_open,
    .release = test_ioctl_close,
    .read = test_ioctl_read,
    .unlocked_ioctl = test_ioctl_ioctl,
};

static int __init ioctl_init(void)
{
    dev_t dev;
    int alloc_ret = -1;
    int cdev_ret = -1;
    alloc_ret = alloc_chrdev_region(&dev, 0, num_of_dev, DRIVER_NAME);

    if (alloc_ret)
        goto error;

    test_ioctl_major = MAJOR(dev);
    cdev_init(&test_ioctl_cdev, &fops);
    cdev_ret = cdev_add(&test_ioctl_cdev, dev, num_of_dev);

    if (cdev_ret)
        goto error;

    pr_alert("%s driver(major: %d) installed.\n", DRIVER_NAME,
             test_ioctl_major);
    return 0;
error:
    if (cdev_ret == 0)
        cdev_del(&test_ioctl_cdev);
    if (alloc_ret == 0)
        unregister_chrdev_region(dev, num_of_dev);
    return -1;
}

static void __exit ioctl_exit(void)
{
    dev_t dev = MKDEV(test_ioctl_major, 0);

    cdev_del(&test_ioctl_cdev);
    unregister_chrdev_region(dev, num_of_dev);
    pr_alert("%s driver removed.\n", DRIVER_NAME);
}

module_init(ioctl_init);
module_exit(ioctl_exit);

MODULE_LICENSE("GPL");
```

```
MODULE_DESCRIPTION("This is test_ioctl module");
```

You can see there is an argument called `cmd` in `test_ioctl_ioctl()` function. It is the ioctl number. The ioctl number encodes the major device number, the type of the ioctl, the command, and the type of the parameter. This ioctl number is usually created by a macro call (`_IO`, `_IOR`, `_IOW` or `_IOWR` — depending on the type) in a header file. This header file should then be included both by the programs which will use ioctl (so they can generate the appropriate ioctl's) and by the kernel module (so it can understand it). In the example below, the header file is `chardev.h` and the program which uses it is `userspace_ioctl.c`.

If you want to use ioctls in your own kernel modules, it is best to receive an official ioctl assignment, so if you accidentally get somebody else's ioctls, or if they get yours, you'll know something is wrong. For more information, consult the kernel source tree at Documentation/userspace-api/ioctl/ioctl-number.rst.

Also, we need to be careful that concurrent access to the shared resources will lead to the race condition. The solution is using atomic Compare-And-Swap (CAS), which we mentioned at 6.5 section, to enforce the exclusive access.

```
/*
 * chardev2.c - Create an input/output character device
 */

#include <linux/atomic.h>
#include <linux/cdev.h>
#include <linux/delay.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/module.h> /* Specifically, a module */
#include <linux/printk.h>
#include <linux/types.h>
#include <linux/uaccess.h> /* for get_user and put_user */
#include <linux/version.h>

#include <asm/errno.h>

#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80

enum {
    CDEV_NOT_USED = 0,
    CDEV_EXCLUSIVE_OPEN = 1,
};

/* Is the device open right now? Used to prevent concurrent access into
 * the same device
 */
static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);

/* The message the device will give when asked */
```

```c
static char message[BUF_LEN + 1];

static struct class *cls;

/* This is called whenever a process attempts to open the device file */
static int device_open(struct inode *inode, struct file *file)
{
    pr_info("device_open(%p)\n", file);

    try_module_get(THIS_MODULE);
    return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
    pr_info("device_release(%p,%p)\n", inode, file);

    module_put(THIS_MODULE);
    return SUCCESS;
}

/* This function is called whenever a process which has already opened the
 * device file attempts to read from it.
 */
static ssize_t device_read(struct file *file, /* see include/linux/fs.h   */
                           char __user *buffer, /* buffer to be filled  */
                           size_t length, /* length of the buffer     */
                           loff_t *offset)
{
    /* Number of bytes actually written to the buffer */
    int bytes_read = 0;
    /* How far did the process reading the message get? Useful if the message
     * is larger than the size of the buffer we get to fill in device_read.
     */
    const char *message_ptr = message;

    if (!*(message_ptr + *offset)) { /* we are at the end of message */
        *offset = 0; /* reset the offset */
        return 0; /* signify end of file */
    }

    message_ptr += *offset;

    /* Actually put the data into the buffer */
    while (length && *message_ptr) {
        /* Because the buffer is in the user data segment, not the kernel
         * data segment, assignment would not work. Instead, we have to
         * use put_user which copies data from the kernel data segment to
         * the user data segment.
         */
        put_user(*(message_ptr++), buffer++);
        length--;
        bytes_read++;
    }

    pr_info("Read %d bytes, %ld left\n", bytes_read, length);

```

```
    *offset += bytes_read;

    /* Read functions are supposed to return the number of bytes actually
     * inserted into the buffer.
     */
    return bytes_read;
}

/* called when somebody tries to write into our device file. */
static ssize_t device_write(struct file *file, const char __user *buffer,
                            size_t length, loff_t *offset)
{
    int i;

    pr_info("device_write(%p,%p,%ld)", file, buffer, length);

    for (i = 0; i < length && i < BUF_LEN; i++)
        get_user(message[i], buffer + i);

    /* Again, return the number of input characters used. */
    return i;
}

/* This function is called whenever a process tries to do an ioctl on our
 * device file. We get two extra parameters (additional to the inode and file
 * structures, which all device functions get): the number of the ioctl called
 * and the parameter given to the ioctl function.
 *
 * If the ioctl is write or read/write (meaning output is returned to the
 * calling process), the ioctl call returns the output of this function.
 */
static long
device_ioctl(struct file *file, /* ditto */
             unsigned int ioctl_num, /* number and param for ioctl */
             unsigned long ioctl_param)
{
    int i;
    long ret = SUCCESS;

    /* We don't want to talk to two processes at the same time. */
    if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
        return -EBUSY;

    /* Switch according to the ioctl called */
    switch (ioctl_num) {
    case IOCTL_SET_MSG: {
        /* Receive a pointer to a message (in user space) and set that to
         * be the device's message. Get the parameter given to ioctl by
         * the process.
         */
        char __user *tmp = (char __user *)ioctl_param;
        char ch;

        /* Find the length of the message */
        get_user(ch, tmp);
        for (i = 0; ch && i < BUF_LEN; i++, tmp++)
```

```
            get_user(ch, tmp);

        device_write(file, (char __user *)ioctl_param, i, NULL);
        break;
    }
    case IOCTL_GET_MSG: {
        loff_t offset = 0;

        /* Give the current message to the calling process - the parameter
         * we got is a pointer, fill it.
         */
        i = device_read(file, (char __user *)ioctl_param, 99, &offset);

        /* Put a zero at the end of the buffer, so it will be properly
         * terminated.
         */
        put_user('\0', (char __user *)ioctl_param + i);
        break;
    }
    case IOCTL_GET_NTH_BYTE:
        /* This ioctl is both input (ioctl_param) and output (the return
         * value of this function).
         */
        ret = (long)message[ioctl_param];
        break;
    }

    /* We're now ready for our next caller */
    atomic_set(&already_open, CDEV_NOT_USED);

    return ret;
}

/* Module Declarations */

/* This structure will hold the functions to be called when a process does
 * something to the device we created. Since a pointer to this structure
 * is kept in the devices table, it can't be local to init_module. NULL is
 * for unimplemented functions.
 */
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .unlocked_ioctl = device_ioctl,
    .open = device_open,
    .release = device_release, /* a.k.a. close */
};

/* Initialize the module - Register the character device */
static int __init chardev2_init(void)
{
    /* Register the character device (atleast try) */
    int ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);

    /* Negative values signify an error */
    if (ret_val < 0) {
        pr_alert("%s failed with %d\n",
```

```
                 "Sorry, registering the character device ", ret_val);
        return ret_val;
    }

#if LINUX_VERSION_CODE >= KERNEL_VERSION(6, 4, 0)
    cls = class_create(DEVICE_FILE_NAME);
#else
    cls = class_create(THIS_MODULE, DEVICE_FILE_NAME);
#endif
    device_create(cls, NULL, MKDEV(MAJOR_NUM, 0), NULL, DEVICE_FILE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_FILE_NAME);

    return 0;
}

/* Cleanup - unregister the appropriate file from /proc */
static void __exit chardev2_exit(void)
{
    device_destroy(cls, MKDEV(MAJOR_NUM, 0));
    class_destroy(cls);

    /* Unregister the device */
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
}

module_init(chardev2_init);
module_exit(chardev2_exit);

MODULE_LICENSE("GPL");
```

```
/*
 * chardev.h - the header file with the ioctl definitions.
 *
 * The declarations here have to be in a header file, because they need
 * to be known both to the kernel module (in chardev2.c) and the process
 * calling ioctl() (in userspace_ioctl.c).
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/* The major device number. We can not rely on dynamic registration
 * any more, because ioctls need to know it.
 */
#define MAJOR_NUM 100

/* Set the message of the device driver */
#define IOCTL_SET_MSG _IOW(MAJOR_NUM, 0, char *)
/* _IOW means that we are creating an ioctl command number for passing
 * information from a user process to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device number we are using.
 *
 * The second argument is the number of the command (there could be several
```

```
 * with different meanings).
 *
 * The third argument is the type we want to get from the process to the
 * kernel.
 */

/* Get the message of the device driver */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/* This IOCTL is used for output, to get the message of the device driver.
 * However, we still need the buffer to place the message in to be input,
 * as it is allocated by the process.
 */

/* Get the n'th byte of the message */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/* The IOCTL is used for both input and output. It receives from the user
 * a number, n, and returns message[n].
 */

/* The name of the device file */
#define DEVICE_FILE_NAME "char_dev"
#define DEVICE_PATH "/dev/char_dev"

#endif
```

```
/*  userspace_ioctl.c - the process to use ioctl's to control the kernel
↪  module
 *
 *  Until now we could have used cat for input and output.  But now
 *  we need to do ioctl's, which require writing our own process.
 */

/* device specifics, such as ioctl numbers and the
 * major device file. */
#include "../chardev.h"

#include <stdio.h> /* standard I/O */
#include <fcntl.h> /* open */
#include <unistd.h> /* close */
#include <stdlib.h> /* exit */
#include <sys/ioctl.h> /* ioctl */

/* Functions for the ioctl calls */

int ioctl_set_msg(int file_desc, char *message)
{
    int ret_val;

    ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

    if (ret_val < 0) {
        printf("ioctl_set_msg failed:%d\n", ret_val);
    }

    return ret_val;
}

```

```c
int ioctl_get_msg(int file_desc)
{
    int ret_val;
    char message[100] = { 0 };

    /* Warning - this is dangerous because we don't tell
   * the kernel how far it's allowed to write, so it
   * might overflow the buffer. In a real production
   * program, we would have used two ioctls - one to tell
   * the kernel the buffer length and another to give
   * it the buffer to fill
   */
    ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

    if (ret_val < 0) {
        printf("ioctl_get_msg failed:%d\n", ret_val);
    }
    printf("get_msg message:%s", message);

    return ret_val;
}

int ioctl_get_nth_byte(int file_desc)
{
    int i, c;

    printf("get_nth_byte message:");

    i = 0;
    do {
        c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

        if (c < 0) {
            printf("\nioctl_get_nth_byte failed at the %d'th byte:\n", i);
            return c;
        }

        putchar(c);
    } while (c != 0);

    return 0;
}

/* Main - Call the ioctl functions */
int main(void)
{
    int file_desc, ret_val;
    char *msg = "Message passed by ioctl\n";

    file_desc = open(DEVICE_PATH, O_RDWR);
    if (file_desc < 0) {
        printf("Can't open device file: %s, error:%d\n", DEVICE_PATH,
               file_desc);
        exit(EXIT_FAILURE);
    }

    ret_val = ioctl_set_msg(file_desc, msg);
```

```c
    if (ret_val)
        goto error;
    ret_val = ioctl_get_nth_byte(file_desc);
    if (ret_val)
        goto error;
    ret_val = ioctl_get_msg(file_desc);
    if (ret_val)
        goto error;

    close(file_desc);
    return 0;
error:
    close(file_desc);
    exit(EXIT_FAILURE);
}
```

10 System Calls

So far, the only thing we've done was to use well defined kernel mechanisms to register `/proc` files and device handlers. This is fine if you want to do something the kernel programmers thought you'd want, such as write a device driver. But what if you want to do something unusual, to change the behavior of the system in some way? Then, you are mostly on your own.

Should one choose not to use a virtual machine, kernel programming can become risky. For example, while writing the code below, the `open()` system call was inadvertently disrupted. This resulted in an inability to open any files, run programs, or shut down the system, necessitating a restart of the virtual machine. Fortunately, no critical files were lost in this instance. However, if such modifications were made on a live, mission-critical system, the consequences could be severe. To mitigate the risk of file loss, even in a test environment, it is advised to execute `sync` right before using `insmod` and `rmmod`.

Forget about `/proc` files, forget about device files. They are just minor details. Minutiae in the vast expanse of the universe. The real process to kernel communication mechanism, the one used by all processes, is *system calls*. When a process requests a service from the kernel (such as opening a file, forking to a new process, or requesting more memory), this is the mechanism used. If you want to change the behaviour of the kernel in interesting ways, this is the place to do it. By the way, if you want to see which system calls a program uses, run `strace <arguments>`.

In general, a process is not supposed to be able to access the kernel. It can not access kernel memory and it can't call kernel functions. The hardware of the CPU enforces this (that is the reason why it is called "protected mode" or "page protection").

System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them). Under

Intel CPUs, this is done by means of interrupt 0x80. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel — and therefore you're allowed to do whatever you want.

The location in the kernel a process can jump to is called `system_call`. The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (`sys_call_table`) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process (or to a different process, if the process time ran out). If you want to read this code, it is at the source file `arch/$(architecture)/kernel/entry.S`, after the line `ENTRY(system_call)`.

So, if we want to change the way a certain system call works, what we need to do is to write our own function to implement it (usually by adding a bit of our own code, and then calling the original function) and then change the pointer at `sys_call_table` to point to our function. Because we might be removed later and we don't want to leave the system in an unstable state, it's important for `cleanup_module` to restore the table to its original state.

To modify the content of `sys_call_table`, we need to consider the control register. A control register is a processor register that changes or controls the general behavior of the CPU. For x86 architecture, the `cr0` register has various control flags that modify the basic operation of the processor. The `WP` flag in `cr0` stands for write protection. Once the `WP` flag is set, the processor disallows further write attempts to the read-only sections Therefore, we must disable the `WP` flag before modifying `sys_call_table`. Since Linux v5.3, the `write_cr0` function cannot be used because of the sensitive `cr0` bits pinned by the security issue, the attacker may write into CPU control registers to disable CPU protections like write protection. As a result, we have to provide the custom assembly routine to bypass it.

However, `sys_call_table` symbol is unexported to prevent misuse. But there have few ways to get the symbol, manual symbol lookup and `kallsyms_lookup_name`. Here we use both depend on the kernel version.

Because of the *control-flow integrity*, which is a technique to prevent the redirect execution code from the attacker, for making sure that the indirect calls go to the expected addresses and the return addresses are not changed. Since Linux v5.7, the kernel patched the series of *control-flow enforcement* (CET) for x86, and some configurations of GCC, like GCC versions 9 and 10 in Ubuntu Linux, will add with CET (the `-fcf-protection` option) in the kernel by default. Using that GCC to compile the kernel with retpoline off may result in CET being enabled in the kernel. You can use the following command to check out the `-fcf-protection` option is enabled or not:

```
$ gcc -v -Q -O2 --help=target | grep protection
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
```

```
...
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
COLLECT_GCC_OPTIONS='-v' '-Q' '-O2' '--help=target' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/9/cc1 -v ... -fcf-protection ...
 GNU C17 (Ubuntu 9.3.0-17ubuntu1~20.04) version 9.3.0 (x86_64-linux-gnu)
...
```

But CET should not be enabled in the kernel, it may break the Kprobes and bpf. Consequently, CET is disabled since v5.11. To guarantee the manual symbol lookup worked, we only use up to v5.4.

Unfortunately, since Linux v5.7 `kallsyms_lookup_name` is also unexported, it needs certain trick to get the address of `kallsyms_lookup_name`. If `CONFIG_KPROBES` is enabled, we can facilitate the retrieval of function addresses by means of Kprobes to dynamically break into the specific kernel routine. Kprobes inserts a breakpoint at the entry of function by replacing the first bytes of the probed instruction. When a CPU hits the breakpoint, registers are stored, and the control will pass to Kprobes. It passes the addresses of the saved registers and the Kprobe struct to the handler you defined, then executes it. Kprobes can be registered by symbol name or address. Within the symbol name, the address will be handled by the kernel.

Otherwise, specify the address of `sys_call_table` from `/proc/kallsyms` and `/boot/System.map` into `sym` parameter. Following is the sample usage for `/proc/kallsyms`:

```
$ sudo grep sys_call_table /proc/kallsyms
ffffffff82000280 R x32_sys_call_table
ffffffff820013a0 R sys_call_table
ffffffff820023e0 R ia32_sys_call_table
$ sudo insmod syscall-steal.ko sym=0xffffffff820013a0
```

Using the address from `/boot/System.map`, be careful about `KASLR` (Kernel Address Space Layout Randomization). `KASLR` may randomize the address of kernel code and data at every boot time, such as the static address listed in `/boot/System.map` will offset by some entropy. The purpose of `KASLR` is to protect the kernel space from the attacker. Without `KASLR`, the attacker may find the target address in the fixed address easily. Then the attacker can use return-oriented programming to insert some malicious codes to execute or receive the target data by a tampered pointer. `KASLR` mitigates these kinds of attacks because the attacker cannot immediately know the target address, but a brute-force attack can still work. If the address of a symbol in `/proc/kallsyms` is different from the address in `/boot/System.map`, `KASLR` is enabled with the kernel, which your system running on.

```
$ grep GRUB_CMDLINE_LINUX_DEFAULT /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
$ sudo grep sys_call_table /boot/System.map-$(uname -r)
ffffffff82000300 R sys_call_table
```

```
$ sudo grep sys_call_table /proc/kallsyms
ffffffff820013a0 R sys_call_table
# Reboot
$ sudo grep sys_call_table /boot/System.map-$(uname -r)
ffffffff82000300 R sys_call_table
$ sudo grep sys_call_table /proc/kallsyms
ffffffff86400300 R sys_call_table
```

If `KASLR` is enabled, we have to take care of the address from `/proc/kallsyms` each time we reboot the machine. In order to use the address from `/boot/System.map`, make sure that `KASLR` is disabled. You can add the `nokaslr` for disabling `KASLR` in next booting time:

```
$ grep GRUB_CMDLINE_LINUX_DEFAULT /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
$ sudo perl -i -pe 'm/quiet/ and s//quiet nokaslr/' /etc/default/grub
$ grep quiet /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="quiet nokaslr splash"
$ sudo update-grub
```

For more information, check out the following:

- Cook: Security things in Linux v5.3
- Unexporting the system call table
- Control-flow integrity for the kernel
- Unexporting kallsyms_lookup_name()
- Kernel Probes (Kprobes)
- Kernel address space layout randomization

The source code here is an example of such a kernel module. We want to "spy" on a certain user, and to `pr_info()` a message whenever that user opens a file. Towards this end, we replace the system call to open a file with our own function, called `our_sys_openat`. This function checks the uid (user's id) of the current process, and if it is equal to the uid we spy on, it calls `pr_info()` to display the name of the file to be opened. Then, either way, it calls the original `openat()` function with the same parameters, to actually open the file.

The `init_module` function replaces the appropriate location in `sys_call_table` and keeps the original pointer in a variable. The `cleanup_module` function uses that variable to restore everything back to normal. This approach is dangerous, because of the possibility of two kernel modules changing the same system call. Imagine we have two kernel modules, A and B. A's openat system call will be `A_openat` and B's will be `B_openat`. Now, when A is inserted into the kernel, the system call is replaced with `A_openat`, which will call the original `sys_openat` when it is done. Next, B is inserted into the kernel, which replaces the system call with `B_openat`, which will call what it thinks is the original system call, `A_openat`, when it's done.

Now, if B is removed first, everything will be well — it will simply restore the system call to `A_openat`, which calls the original. However, if A is removed and then B is removed, the system will crash. A's removal will restore the system call to the original, `sys_openat`, cutting B out of the loop. Then, when B is removed, it will restore the system call to what it thinks is the original, `A_openat`, which is no longer in memory. At first glance, it appears we could solve this particular problem by checking if the system call is equal to our open function and if so not changing it at all (so that B won't change the system call when it is removed), but that will cause an even worse problem. When A is removed, it sees that the system call was changed to `B_openat` so that it is no longer pointing to `A_openat`, so it will not restore it to `sys_openat` before it is removed from memory. Unfortunately, `B_openat` will still try to call `A_openat` which is no longer there, so that even without removing B the system would crash.

For x86 architecture, the system call table cannot be used to invoke a system call after commit 1e3ad78 since v6.9. This commit has been backported to long term stable kernels, like v5.15.154+, v6.1.85+, v6.6.26+ and v6.8.5+, see this answer for more details. In this case, thanks to Kprobes, a hook can be used instead on the system call entry to intercept the system call.

Note that all the related problems make syscall stealing unfeasible for production use. In order to keep people from doing potential harmful things `sys_call_table` is no longer exported. This means, if you want to do something more than a mere dry run of this example, you will have to patch your current kernel in order to have `sys_call_table` exported.

```c
/*
 * syscall-steal.c
 *
 * System call "stealing" sample.
 *
 * Disables page protection at a processor level by changing the 16th bit
 * in the cr0 register (could be Intel specific).
 */

#include <linux/delay.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h> /* which will have params */
#include <linux/unistd.h> /* The list of system calls */
#include <linux/cred.h> /* For current_uid() */
#include <linux/uidgid.h> /* For __kuid_val() */
#include <linux/version.h>

/* For the current (process) structure, we need this to know who the
 * current user is.
 */
#include <linux/sched.h>
#include <linux/uaccess.h>

```

```c
/* The way we access "sys_call_table" varies as kernel internal changes.
 * - Prior to v5.4 : manual symbol lookup
 * - v5.5 to v5.6  : use kallsyms_lookup_name()
 * - v5.7+         : Kprobes or specific kernel module parameter
 */

/* The in-kernel calls to the ksys_close() syscall were removed in Linux v5.11+.
 */
#if (LINUX_VERSION_CODE < KERNEL_VERSION(5, 7, 0))

#if LINUX_VERSION_CODE <= KERNEL_VERSION(5, 4, 0)
#define HAVE_KSYS_CLOSE 1
#include <linux/syscalls.h> /* For ksys_close() */
#else
#include <linux/kallsyms.h> /* For kallsyms_lookup_name */
#endif

#else

#if defined(CONFIG_KPROBES)
#define HAVE_KPROBES 1
#if defined(CONFIG_X86_64)
/* If you have tried to use the syscall table to intercept syscalls and it
 * doesn't work, you can try to use Kprobes to intercept syscalls.
 * Set USE_KPROBES_PRE_HANDLER_BEFORE_SYSCALL to 1 to register a pre-handler
 * before the syscall.
 */
#define USE_KPROBES_PRE_HANDLER_BEFORE_SYSCALL 0
#endif
#include <linux/kprobes.h>
#else
#define HAVE_PARAM 1
#include <linux/kallsyms.h> /* For sprint_symbol */
/* The address of the sys_call_table, which can be obtained with looking up
 * "/boot/System.map" or "/proc/kallsyms". When the kernel version is v5.7+,
 * without CONFIG_KPROBES, you can input the parameter or the module will look
 * up all the memory.
 */
static unsigned long sym = 0;
module_param(sym, ulong, 0644);
#endif /* CONFIG_KPROBES */

#endif /* Version < v5.7 */

/* UID we want to spy on - will be filled from the command line. */
static uid_t uid = -1;
module_param(uid, int, 0644);

#if USE_KPROBES_PRE_HANDLER_BEFORE_SYSCALL

/* syscall_sym is the symbol name of the syscall to spy on. The default is
 * "__x64_sys_openat", which can be changed by the module parameter. You can
 * look up the symbol name of a syscall in /proc/kallsyms.
 */
static char *syscall_sym = "__x64_sys_openat";
module_param(syscall_sym, charp, 0644);
```

```

static int sys_call_kprobe_pre_handler(struct kprobe *p, struct pt_regs *regs)
{
    if (__kuid_val(current_uid()) != uid) {
        return 0;
    }

    pr_info("%s called by %d\n", syscall_sym, uid);
    return 0;
}

static struct kprobe syscall_kprobe = {
    .symbol_name = "__x64_sys_openat",
    .pre_handler = sys_call_kprobe_pre_handler,
};
#else

static unsigned long **sys_call_table_stolen;

/* A pointer to the original system call. The reason we keep this, rather
 * than call the original function (sys_openat), is because somebody else
 * might have replaced the system call before us. Note that this is not
 * 100% safe, because if another module replaced sys_openat before us,
 * then when we are inserted, we will call the function in that module -
 * and it might be removed before we are.
 *
 * Another reason for this is that we can not get sys_openat.
 * It is a static variable, so it is not exported.
 */
#ifdef CONFIG_ARCH_HAS_SYSCALL_WRAPPER
static asmlinkage long (*original_call)(const struct pt_regs *);
#else
static asmlinkage long (*original_call)(int, const char __user *, int,
↪  umode_t);
#endif

/* The function we will replace sys_openat (the function called when you
 * call the open system call) with. To find the exact prototype, with
 * the number and type of arguments, we find the original function first
 * (it is at fs/open.c).
 *
 * In theory, this means that we are tied to the current version of the
 * kernel. In practice, the system calls almost never change (it would
 * wreck havoc and require programs to be recompiled, since the system
 * calls are the interface between the kernel and the processes).
 */
#ifdef CONFIG_ARCH_HAS_SYSCALL_WRAPPER
static asmlinkage long our_sys_openat(const struct pt_regs *regs)
#else
static asmlinkage long our_sys_openat(int dfd, const char __user *filename,
                                      int flags, umode_t mode)
#endif
{
    int i = 0;
    char ch;

    if (__kuid_val(current_uid()) != uid)
```

```
        goto orig_call;

    /* Report the file, if relevant */
    pr_info("Opened file by %d: ", uid);
    do {
#ifdef CONFIG_ARCH_HAS_SYSCALL_WRAPPER
        get_user(ch, (char __user *)regs->si + i);
#else
        get_user(ch, (char __user *)filename + i);
#endif
        i++;
        pr_info("%c", ch);
    } while (ch != 0);
    pr_info("\n");

orig_call:
    /* Call the original sys_openat - otherwise, we lose the ability to
     * open files.
     */
#ifdef CONFIG_ARCH_HAS_SYSCALL_WRAPPER
    return original_call(regs);
#else
    return original_call(dfd, filename, flags, mode);
#endif
}

static unsigned long **acquire_sys_call_table(void)
{
#ifdef HAVE_KSYS_CLOSE
    unsigned long int offset = PAGE_OFFSET;
    unsigned long **sct;

    while (offset < ULLONG_MAX) {
        sct = (unsigned long **)offset;

        if (sct[__NR_close] == (unsigned long *)ksys_close)
            return sct;

        offset += sizeof(void *);
    }

    return NULL;
#endif

#ifdef HAVE_PARAM
    const char sct_name[15] = "sys_call_table";
    char symbol[40] = { 0 };

    if (sym == 0) {
        pr_alert("For Linux v5.7+, Kprobes is the preferable way to get "
                 "symbol.\n");
        pr_info("If Kprobes is absent, you have to specify the address of "
                "sys_call_table symbol\n");
        pr_info("by /boot/System.map or /proc/kallsyms, which contains all the "
                "symbol addresses, into sym parameter.\n");
        return NULL;
```

```c
    }
    sprint_symbol(symbol, sym);
    if (!strncmp(sct_name, symbol, sizeof(sct_name) - 1))
        return (unsigned long **)sym;

    return NULL;
#endif

#ifdef HAVE_KPROBES
    unsigned long (*kallsyms_lookup_name)(const char *name);
    struct kprobe kp = {
        .symbol_name = "kallsyms_lookup_name",
    };

    if (register_kprobe(&kp) < 0)
        return NULL;
    kallsyms_lookup_name = (unsigned long (*)(const char *name))kp.addr;
    unregister_kprobe(&kp);
#endif

    return (unsigned long **)kallsyms_lookup_name("sys_call_table");
}

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 3, 0)
static inline void __write_cr0(unsigned long cr0)
{
    asm volatile("mov %0,%%cr0" : "+r"(cr0) : : "memory");
}
#else
#define __write_cr0 write_cr0
#endif

static void enable_write_protection(void)
{
    unsigned long cr0 = read_cr0();
    set_bit(16, &cr0);
    __write_cr0(cr0);
}

static void disable_write_protection(void)
{
    unsigned long cr0 = read_cr0();
    clear_bit(16, &cr0);
    __write_cr0(cr0);
}
#endif

static int __init syscall_steal_start(void)
{
#if USE_KPROBES_PRE_HANDLER_BEFORE_SYSCALL

    int err;
    /* use symbol name from the module parameter */
    syscall_kprobe.symbol_name = syscall_sym;
    err = register_kprobe(&syscall_kprobe);
    if (err) {
        pr_err("register_kprobe() on %s failed: %d\n", syscall_sym, err);
```

```
        pr_err("Please check the symbol name from 'syscall_sym' parameter.\n");
        return err;
    }

#else
    if (!(sys_call_table_stolen = acquire_sys_call_table()))
        return -1;

    disable_write_protection();

    /* keep track of the original open function */
    original_call = (void *)sys_call_table_stolen[__NR_openat];

    /* use our openat function instead */
    sys_call_table_stolen[__NR_openat] = (unsigned long *)our_sys_openat;

    enable_write_protection();

#endif

    pr_info("Spying on UID:%d\n", uid);
    return 0;
}

static void __exit syscall_steal_end(void)
{
#if USE_KPROBES_PRE_HANDLER_BEFORE_SYSCALL
    unregister_kprobe(&syscall_kprobe);
#else
    if (!sys_call_table_stolen)
        return;

    /* Return the system call back to normal */
    if (sys_call_table_stolen[__NR_openat] != (unsigned long *)our_sys_openat) {
        pr_alert("Somebody else also played with the ");
        pr_alert("open system call\n");
        pr_alert("The system may be left in ");
        pr_alert("an unstable state.\n");
    }

    disable_write_protection();
    sys_call_table_stolen[__NR_openat] = (unsigned long *)original_call;
    enable_write_protection();
#endif

    msleep(2000);
}

module_init(syscall_steal_start);
module_exit(syscall_steal_end);

MODULE_LICENSE("GPL");
```

11 Blocking Processes and threads

11.1 Sleep

What do you do when somebody asks you for something you can not do right away? If you are a human being and you are bothered by a human being, the only thing you can say is: "*Not right now, I'm busy. Go away!*". But if you are a kernel module and you are bothered by a process, you have another possibility. You can put the process to sleep until you can service it. After all, processes are being put to sleep by the kernel and woken up all the time (that is the way multiple processes appear to run on the same time on a single CPU).

This kernel module is an example of this. The file (called `/proc/sleep`) can only be opened by a single process at a time. If the file is already open, the kernel module calls `wait_event_interruptible`. The easiest way to keep a file open is to open it with:

```
tail -f
```

This function changes the status of the task (a task is the kernel data structure which holds information about a process and the system call it is in, if any) to `TASK_INTERRUPTIBLE`, which means that the task will not run until it is woken up somehow, and adds it to WaitQ, the queue of tasks waiting to access the file. Then, the function calls the scheduler to context switch to a different process, one which has some use for the CPU.

When a process is done with the file, it closes it, and `module_close` is called. That function wakes up all the processes in the queue (there's no mechanism to only wake up one of them). It then returns and the process which just closed the file can continue to run. In time, the scheduler decides that that process has had enough and gives control of the CPU to another process. Eventually, one of the processes which was in the queue will be given control of the CPU by the scheduler. It starts at the point right after the call to `wait_event_interruptible`.

This means that the process is still in kernel mode - as far as the process is concerned, it issued the open system call and the system call has not returned yet. The process does not know somebody else used the CPU for most of the time between the moment it issued the call and the moment it returned.

It can then proceed to set a global variable to tell all the other processes that the file is still open and go on with its life. When the other processes get a piece of the CPU, they'll see that global variable and go back to sleep.

So we will use `tail -f` to keep the file open in the background, while trying to access it with another process (again in the background, so that we need not switch to a different vt). As soon as the first background process is killed with kill %1 , the second is woken up, is able to access the file and finally terminates.

To make our life more interesting, `module_close` does not have a monopoly on waking up the processes which wait to access the file. A signal, such as *Ctrl +c* (**SIGINT**) can also wake up a process. This is because we used wait_event_interruptible. We could have used wait_event instead, but that would have resulted in extremely angry users whose *Ctrl+c*'s are ignored.

In that case, we want to return with -EINTR immediately. This is important so users can, for example, kill the process before it receives the file.

There is one more point to remember. Some times processes don't want to sleep, they want either to get what they want immediately, or to be told it cannot be done. Such processes use the O_NONBLOCK flag when opening the file. The kernel is supposed to respond by returning with the error code -EAGAIN from operations which would otherwise block, such as opening the file in this example. The program cat_nonblock, available in the examples/other directory, can be used to open a file with O_NONBLOCK.

```
$ sudo insmod sleep.ko
$ cat_nonblock /proc/sleep
Last input:
$ tail -f /proc/sleep &
Last input:
Last input:
Last input:
Last input:
Last input:
Last input:
Last input:
tail: /proc/sleep: file truncated
[1] 6540
$ cat_nonblock /proc/sleep
Open would block
$ kill %1
[1]+  Terminated              tail -f /proc/sleep
$ cat_nonblock /proc/sleep
Last input:
$
```

```c
/*
 * sleep.c - create a /proc file, and if several processes try to open it
 * at the same time, put all but one to sleep.
 */

#include <linux/atomic.h>
#include <linux/fs.h>
#include <linux/kernel.h> /* for sprintf() */
#include <linux/module.h> /* Specifically, a module */
#include <linux/printk.h>
#include <linux/proc_fs.h> /* Necessary because we use proc fs */
#include <linux/types.h>
#include <linux/uaccess.h> /* for get_user and put_user */
#include <linux/version.h>
#include <linux/wait.h> /* For putting processes to sleep and
                                  waking them up */
```

```c

#include <asm/current.h>
#include <asm/errno.h>

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

/* Here we keep the last message received, to prove that we can process our
 * input.
 */
#define MESSAGE_LENGTH 80
static char message[MESSAGE_LENGTH];

static struct proc_dir_entry *our_proc_file;
#define PROC_ENTRY_FILENAME "sleep"

/* Since we use the file operations struct, we can't use the special proc
 * output provisions - we have to use a standard read function, which is this
 * function.
 */
static ssize_t module_output(struct file *file, /* see include/linux/fs.h   */
                             char __user *buf, /* The buffer to put data to
                                                  (in the user segment)    */
                             size_t len, /* The length of the buffer */
                             loff_t *offset)
{
    static int finished = 0;
    int i;
    char output_msg[MESSAGE_LENGTH + 30];

    /* Return 0 to signify end of file - that we have nothing more to say
     * at this point.
     */
    if (finished) {
        finished = 0;
        return 0;
    }

    sprintf(output_msg, "Last input:%s\n", message);
    for (i = 0; i < len && output_msg[i]; i++)
        put_user(output_msg[i], buf + i);

    finished = 1;
    return i; /* Return the number of bytes "read" */
}

/* This function receives input from the user when the user writes to the
 * /proc file.
 */
static ssize_t module_input(struct file *file, /* The file itself */
                            const char __user *buf, /* The buffer with input */
                            size_t length, /* The buffer's length */
                            loff_t *offset) /* offset to file - ignore */
{
    int i;
```

```

    /* Put the input into Message, where module_output will later be able
     * to use it.
     */
    for (i = 0; i < MESSAGE_LENGTH - 1 && i < length; i++)
        get_user(message[i], buf + i);
    /* we want a standard, zero terminated string */
    message[i] = '\0';

    /* We need to return the number of input characters used */
    return i;
}

/* 1 if the file is currently open by somebody */
static atomic_t already_open = ATOMIC_INIT(0);

/* Queue of processes who want our file */
static DECLARE_WAIT_QUEUE_HEAD(waitq);

/* Called when the /proc file is opened */
static int module_open(struct inode *inode, struct file *file)
{
    /* Try to get without blocking  */
    if (!atomic_cmpxchg(&already_open, 0, 1)) {
        /* Success without blocking, allow the access */
        try_module_get(THIS_MODULE);
        return 0;
    }
    /* If the file's flags include O_NONBLOCK, it means the process does not
     * want to wait for the file. In this case, because the file is already
↪    open,
     * we should fail with -EAGAIN, meaning "you will have to try again",
     * instead of blocking a process which would rather stay awake.
     */
    if (file->f_flags & O_NONBLOCK)
        return -EAGAIN;

    /* This is the correct place for try_module_get(THIS_MODULE) because if
     * a process is in the loop, which is within the kernel module,
     * the kernel module must not be removed.
     */
    try_module_get(THIS_MODULE);

    while (atomic_cmpxchg(&already_open, 0, 1)) {
        int i, is_sig = 0;

        /* This function puts the current process, including any system
         * calls, such as us, to sleep.  Execution will be resumed right
         * after the function call, either because somebody called
         * wake_up(&waitq) (only module_close does that, when the file
         * is closed) or when a signal, such as Ctrl-C, is sent
         * to the process
         */
        wait_event_interruptible(waitq, !atomic_read(&already_open));

        /* If we woke up because we got a signal we're not blocking,
         * return -EINTR (fail the system call).  This allows processes
```

```
         * to be killed or stopped.
         */
        for (i = 0; i < _NSIG_WORDS && !is_sig; i++)
            is_sig = current->pending.signal.sig[i] &
            ↪  ~current->blocked.sig[i];

        if (is_sig) {
            /* It is important to put module_put(THIS_MODULE) here, because
             * for processes where the open is interrupted there will never
             * be a corresponding close. If we do not decrement the usage
             * count here, we will be left with a positive usage count
             * which we will have no way to bring down to zero, giving us
             * an immortal module, which can only be killed by rebooting
             * the machine.
             */
            module_put(THIS_MODULE);
            return -EINTR;
        }
    }

    return 0; /* Allow the access */
}

/* Called when the /proc file is closed */
static int module_close(struct inode *inode, struct file *file)
{
    /* Set already_open to zero, so one of the processes in the waitq will
     * be able to set already_open back to one and to open the file. All
     * the other processes will be called when already_open is back to one,
     * so they'll go back to sleep.
     */
    atomic_set(&already_open, 0);

    /* Wake up all the processes in waitq, so if anybody is waiting for the
     * file, they can have it.
     */
    wake_up(&waitq);

    module_put(THIS_MODULE);

    return 0; /* success */
}

/* Structures to register as the /proc file, with pointers to all the relevant
 * functions.
 */

/* File operations for our proc file. This is where we place pointers to all
 * the functions called when somebody tries to do something to our file. NULL
 * means we don't want to deal with something.
 */
#ifdef HAVE_PROC_OPS
static const struct proc_ops file_ops_4_our_proc_file = {
    .proc_read = module_output, /* "read" from the file */
    .proc_write = module_input, /* "write" to the file */
    .proc_open = module_open, /* called when the /proc file is opened */
    .proc_release = module_close, /* called when it's closed */
```

```
    .proc_lseek = noop_llseek, /* return file->f_pos */
};
#else
static const struct file_operations file_ops_4_our_proc_file = {
    .read = module_output,
    .write = module_input,
    .open = module_open,
    .release = module_close,
    .llseek = noop_llseek,
};
#endif

/* Initialize the module - register the proc file */
static int __init sleep_init(void)
{
    our_proc_file =
        proc_create(PROC_ENTRY_FILENAME, 0644, NULL, &file_ops_4_our_proc_file);
    if (our_proc_file == NULL) {
        pr_debug("Error: Could not initialize /proc/%s\n", PROC_ENTRY_FILENAME);
        return -ENOMEM;
    }
    proc_set_size(our_proc_file, 80);
    proc_set_user(our_proc_file, GLOBAL_ROOT_UID, GLOBAL_ROOT_GID);

    pr_info("/proc/%s created\n", PROC_ENTRY_FILENAME);

    return 0;
}

/* Cleanup - unregister our file from /proc.  This could get dangerous if
 * there are still processes waiting in waitq, because they are inside our
 * open function, which will get unloaded. I'll explain how to avoid removal
 * of a kernel module in such a case in chapter 10.
 */
static void __exit sleep_exit(void)
{
    remove_proc_entry(PROC_ENTRY_FILENAME, NULL);
    pr_debug("/proc/%s removed\n", PROC_ENTRY_FILENAME);
}

module_init(sleep_init);
module_exit(sleep_exit);

MODULE_LICENSE("GPL");
```

```
/*
 *  cat_nonblock.c - open a file and display its contents, but exit rather than
 *  wait for input.
 */
#include <errno.h> /* for errno */
#include <fcntl.h> /* for open */
#include <stdio.h> /* standard I/O */
#include <stdlib.h> /* for exit */
#include <unistd.h> /* for read */
```

```c

#define MAX_BYTES 1024 * 4

int main(int argc, char *argv[])
{
    int fd; /* The file descriptor for the file to read */
    size_t bytes; /* The number of bytes read */
    char buffer[MAX_BYTES]; /* The buffer for the bytes */

    /* Usage */
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        puts("Reads the content of a file, but doesn't wait for input");
        exit(-1);
    }

    /* Open the file for reading in non blocking mode */
    fd = open(argv[1], O_RDONLY | O_NONBLOCK);

    /* If open failed */
    if (fd == -1) {
        puts(errno == EAGAIN ? "Open would block" : "Open failed");
        exit(-1);
    }

    /* Read the file and output its contents */
    do {
        /* Read characters from the file */
        bytes = read(fd, buffer, MAX_BYTES);

        /* If there's an error, report it and die */
        if (bytes == -1) {
            if (errno == EAGAIN)
                puts("Normally I'd block, but you told me not to");
            else
                puts("Another read error");
            exit(-1);
        }

        /* Print the characters */
        if (bytes > 0) {
            for (int i = 0; i < bytes; i++)
                putchar(buffer[i]);
        }

        /* While there are no errors and the file isn't over */
    } while (bytes > 0);

    return 0;
}
```

11.2 Completions

Sometimes one thing should happen before another within a module having multiple threads. Rather than using `/bin/sleep` commands, the kernel has another way to do this which allows timeouts or interrupts to also happen.

Completions as code synchronization mechanism have three main parts, initialization of struct completion synchronization object, the waiting or barrier part through `wait_for_completion()`, and the signalling side through a call to `complete()`.

In the subsequent example, two threads are initiated: crank and flywheel. It is imperative that the crank thread starts before the flywheel thread. A completion state is established for each of these threads, with a distinct completion defined for both the crank and flywheel threads. At the exit point of each thread the respective completion state is updated, and `wait_for_completion` is used by the flywheel thread to ensure that it does not begin prematurely. The crank thread uses the `complete_all()` function to update the completion, which lets the flywheel thread continue.

So even though `flywheel_thread` is started first you should notice when you load this module and run `dmesg`, that turning the crank always happens first because the flywheel thread waits for the crank thread to complete.

There are other variations of the `wait_for_completion` function, which include timeouts or being interrupted, but this basic mechanism is enough for many common situations without adding a lot of complexity.

```
/*
 * completions.c
 */
#include <linux/completion.h>
#include <linux/err.h> /* for IS_ERR() */
#include <linux/init.h>
#include <linux/kthread.h>
#include <linux/module.h>
#include <linux/printk.h>
#include <linux/version.h>

static struct completion crank_comp;
static struct completion flywheel_comp;

static int machine_crank_thread(void *arg)
{
    pr_info("Turn the crank\n");

    complete_all(&crank_comp);
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 17, 0)
    kthread_complete_and_exit(&crank_comp, 0);
#else
    complete_and_exit(&crank_comp, 0);
#endif
}

static int machine_flywheel_spinup_thread(void *arg)
{
    wait_for_completion(&crank_comp);

    pr_info("Flywheel spins up\n");

```

```c
    complete_all(&flywheel_comp);
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 17, 0)
    kthread_complete_and_exit(&flywheel_comp, 0);
#else
    complete_and_exit(&flywheel_comp, 0);
#endif
}

static int __init completions_init(void)
{
    struct task_struct *crank_thread;
    struct task_struct *flywheel_thread;

    pr_info("completions example\n");

    init_completion(&crank_comp);
    init_completion(&flywheel_comp);

    crank_thread = kthread_create(machine_crank_thread, NULL, "KThread Crank");
    if (IS_ERR(crank_thread))
        goto ERROR_THREAD_1;

    flywheel_thread = kthread_create(machine_flywheel_spinup_thread, NULL,
                                     "KThread Flywheel");
    if (IS_ERR(flywheel_thread))
        goto ERROR_THREAD_2;

    wake_up_process(flywheel_thread);
    wake_up_process(crank_thread);

    return 0;

ERROR_THREAD_2:
    kthread_stop(crank_thread);
ERROR_THREAD_1:

    return -1;
}

static void __exit completions_exit(void)
{
    wait_for_completion(&crank_comp);
    wait_for_completion(&flywheel_comp);

    pr_info("completions exit\n");
}

module_init(completions_init);
module_exit(completions_exit);

MODULE_DESCRIPTION("Completions example");
MODULE_LICENSE("GPL");
```

12 Avoiding Collisions and Deadlocks

If processes running on different CPUs or in different threads try to access the same memory, then it is possible that strange things can happen or your system can lock up. To avoid this, various types of mutual exclusion kernel functions are available. These indicate if a section of code is "locked" or "unlocked" so that simultaneous attempts to run it can not happen.

12.1 Mutex

You can use kernel mutexes (mutual exclusions) in much the same manner that you might deploy them in userland. This may be all that is needed to avoid collisions in most cases.

```
/*
 * example_mutex.c
 */
#include <linux/module.h>
#include <linux/mutex.h>
#include <linux/printk.h>

static DEFINE_MUTEX(mymutex);

static int __init example_mutex_init(void)
{
    int ret;

    pr_info("example_mutex init\n");

    ret = mutex_trylock(&mymutex);
    if (ret != 0) {
        pr_info("mutex is locked\n");

        if (mutex_is_locked(&mymutex) == 0)
            pr_info("The mutex failed to lock!\n");

        mutex_unlock(&mymutex);
        pr_info("mutex is unlocked\n");
    } else
        pr_info("Failed to lock\n");

    return 0;
}

static void __exit example_mutex_exit(void)
{
    pr_info("example_mutex exit\n");
}

module_init(example_mutex_init);
module_exit(example_mutex_exit);

MODULE_DESCRIPTION("Mutex example");
MODULE_LICENSE("GPL");
```

12.2 Spinlocks

As the name suggests, spinlocks lock up the CPU that the code is running on, taking 100% of its resources. Because of this you should only use the spinlock mechanism around code which is likely to take no more than a few milliseconds to run and so will not noticeably slow anything down from the user's point of view.

The example here is `"irq safe"` in that if interrupts happen during the lock then they will not be forgotten and will activate when the unlock happens, using the `flags` variable to retain their state.

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

static DEFINE_SPINLOCK(sl_static);
static spinlock_t sl_dynamic;

static void example_spinlock_static(void)
{
    unsigned long flags;

    spin_lock_irqsave(&sl_static, flags);
    pr_info("Locked static spinlock\n");

    /* Do something or other safely. Because this uses 100% CPU time, this
     * code should take no more than a few milliseconds to run.
     */

    spin_unlock_irqrestore(&sl_static, flags);
    pr_info("Unlocked static spinlock\n");
}

static void example_spinlock_dynamic(void)
{
    unsigned long flags;

    spin_lock_init(&sl_dynamic);
    spin_lock_irqsave(&sl_dynamic, flags);
    pr_info("Locked dynamic spinlock\n");

    /* Do something or other safely. Because this uses 100% CPU time, this
     * code should take no more than a few milliseconds to run.
     */

    spin_unlock_irqrestore(&sl_dynamic, flags);
    pr_info("Unlocked dynamic spinlock\n");
}

static int __init example_spinlock_init(void)
{
```

```c
    pr_info("example spinlock started\n");

    example_spinlock_static();
    example_spinlock_dynamic();

    return 0;
}

static void __exit example_spinlock_exit(void)
{
    pr_info("example spinlock exit\n");
}

module_init(example_spinlock_init);
module_exit(example_spinlock_exit);

MODULE_DESCRIPTION("Spinlock example");
MODULE_LICENSE("GPL");
```

Taking 100% of a CPU's resources comes with greater responsibility. Situations where the kernel code monopolizes a CPU are called **atomic contexts**. Holding a spinlock is one of those situations. Sleeping in atomic contexts may leave the system hanging, as the occupied CPU devotes 100% of its resources doing nothing but sleeping. In some worse cases the system may crash. Thus, sleeping in atomic contexts is considered a bug in the kernel. They are sometimes called "sleep-in-atomic-context" in some materials.

Note that sleeping here is not limited to calling the sleep functions explicitly. If subsequent function calls eventually invoke a function that sleeps, it is also considered sleeping. Thus, it is important to pay attention to functions being used in atomic context. There's no documentation recording all such functions, but code comments may help. Sometimes you may find comments in kernel source code stating that a function "may sleep", "might sleep", or more explicitly "the caller should not hold a spinlock". Those comments are hints that a function may implicitly sleep and must not be called in atomic contexts.

12.3 Read and write locks

Read and write locks are specialised kinds of spinlocks so that you can exclusively read from something or write to something. Like the earlier spinlocks example, the one below shows an "irq safe" situation in which if other functions were triggered from irqs which might also read and write to whatever you are concerned with then they would not disrupt the logic. As before it is a good idea to keep anything done within the lock as short as possible so that it does not hang up the system and cause users to start revolting against the tyranny of your module.

```c
/*
 * example_rwlock.c
 */
```

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

static DEFINE_RWLOCK(myrwlock);

static void example_read_lock(void)
{
    unsigned long flags;

    read_lock_irqsave(&myrwlock, flags);
    pr_info("Read Locked\n");

    /* Read from something */

    read_unlock_irqrestore(&myrwlock, flags);
    pr_info("Read Unlocked\n");
}

static void example_write_lock(void)
{
    unsigned long flags;

    write_lock_irqsave(&myrwlock, flags);
    pr_info("Write Locked\n");

    /* Write to something */

    write_unlock_irqrestore(&myrwlock, flags);
    pr_info("Write Unlocked\n");
}

static int __init example_rwlock_init(void)
{
    pr_info("example_rwlock started\n");

    example_read_lock();
    example_write_lock();

    return 0;
}

static void __exit example_rwlock_exit(void)
{
    pr_info("example_rwlock exit\n");
}

module_init(example_rwlock_init);
module_exit(example_rwlock_exit);

MODULE_DESCRIPTION("Read/Write locks example");
MODULE_LICENSE("GPL");
```

Of course, if you know for sure that there are no functions triggered by irqs which could possibly interfere with your logic then you can use the simpler `read_lock(&myrwlock)` and `read_unlock(&myrwlock)` or the corresponding write functions.

12.4 Atomic operations

If you are doing simple arithmetic: adding, subtracting or bitwise operations, then there is another way in the multi-CPU and multi-hyperthreaded world to stop other parts of the system from messing with your mojo. By using atomic operations you can be confident that your addition, subtraction or bit flip did actually happen and was not overwritten by some other shenanigans. An example is shown below.

```c
/*
 * example_atomic.c
 */
#include <linux/atomic.h>
#include <linux/bitops.h>
#include <linux/module.h>
#include <linux/printk.h>

#define BYTE_TO_BINARY_PATTERN "%c%c%c%c%c%c%c%c"
#define BYTE_TO_BINARY(byte) \
    ((byte & 0x80) ? '1' : '0'), ((byte & 0x40) ? '1' : '0'), \
        ((byte & 0x20) ? '1' : '0'), ((byte & 0x10) ? '1' : '0'), \
        ((byte & 0x08) ? '1' : '0'), ((byte & 0x04) ? '1' : '0'), \
        ((byte & 0x02) ? '1' : '0'), ((byte & 0x01) ? '1' : '0')

static void atomic_add_subtract(void)
{
    atomic_t debbie;
    atomic_t chris = ATOMIC_INIT(50);

    atomic_set(&debbie, 45);

    /* subtract one */
    atomic_dec(&debbie);

    atomic_add(7, &debbie);

    /* add one */
    atomic_inc(&debbie);

    pr_info("chris: %d, debbie: %d\n", atomic_read(&chris),
            atomic_read(&debbie));
}

static void atomic_bitwise(void)
{
    unsigned long word = 0;

    pr_info("Bits 0: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    set_bit(3, &word);
```

```c
    set_bit(5, &word);
    pr_info("Bits 1: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    clear_bit(5, &word);
    pr_info("Bits 2: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    change_bit(3, &word);

    pr_info("Bits 3: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));
    if (test_and_set_bit(3, &word))
        pr_info("wrong\n");
    pr_info("Bits 4: " BYTE_TO_BINARY_PATTERN, BYTE_TO_BINARY(word));

    word = 255;
    pr_info("Bits 5: " BYTE_TO_BINARY_PATTERN "\n", BYTE_TO_BINARY(word));
}

static int __init example_atomic_init(void)
{
    pr_info("example_atomic started\n");

    atomic_add_subtract();
    atomic_bitwise();

    return 0;
}

static void __exit example_atomic_exit(void)
{
    pr_info("example_atomic exit\n");
}

module_init(example_atomic_init);
module_exit(example_atomic_exit);

MODULE_DESCRIPTION("Atomic operations example");
MODULE_LICENSE("GPL");
```

Before the C11 standard adopts the built-in atomic types, the kernel already provided a small set of atomic types by using a bunch of tricky architecture-specific codes. Implementing the atomic types by C11 atomics may allow the kernel to throw away the architecture-specific codes and letting the kernel code be more friendly to the people who understand the standard. But there are some problems, such as the memory model of the kernel doesn't match the model formed by the C11 atomics. For further details, see:

- kernel documentation of atomic types
- Time to move to C11 atomics?
- Atomic usage patterns in the kernel

13 Replacing Print Macros

13.1 Replacement

In Section 1.7, it was noted that the X Window System and kernel module programming are not conducive to integration. This remains valid during the development of kernel modules. However, in practical scenarios, the necessity emerges to relay messages to the tty (teletype) originating the module load command.

The term “tty” originates from *teletype*, which initially referred to a combined keyboard-printer for Unix system communication. Today, it signifies a text stream abstraction employed by Unix programs, encompassing physical terminals, xterms in X displays, and network connections like SSH.

To achieve this, the “current” pointer is leveraged to access the active task’s tty structure. Within this structure lies a pointer to a string write function, facilitating the string’s transmission to the tty.

```
/*
 * print_string.c - Send output to the tty we're running on, regardless if
 * it is through X11, telnet, etc.  We do this by printing the string to the
 * tty associated with the current task.
 */
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h> /* For current */
#include <linux/tty.h> /* For the tty declarations */

static void print_string(char *str)
{
    /* The tty for the current task */
    struct tty_struct *my_tty = get_current_tty();

    /* If my_tty is NULL, the current task has no tty you can print to (i.e.,
     * if it is a daemon). If so, there is nothing we can do.
     */
    if (my_tty) {
        const struct tty_operations *ttyops = my_tty->driver->ops;
        /* my_tty->driver is a struct which holds the tty's functions,
         * one of which (write) is used to write strings to the tty.
         * It can be used to take a string either from the user's or
         * kernel's memory segment.
         *
         * The function's 1st parameter is the tty to write to, because the
         * same function would normally be used for all tty's of a certain
         * type.
         * The 2nd parameter is a pointer to a string.
         * The 3rd parameter is the length of the string.
         *
         * As you will see below, sometimes it's necessary to use
         * preprocessor stuff to create code that works for different
         * kernel versions. The (naive) approach we've taken here does not
         * scale well. The right way to deal with this is described in
         * section 2 of
```

```
         * linux/Documentation/SubmittingPatches
         */
        (ttyops->write)(my_tty, /* The tty itself */
                        str, /* String */
                        strlen(str)); /* Length */

        /* ttys were originally hardware devices, which (usually) strictly
         * followed the ASCII standard. In ASCII, to move to a new line you
         * need two characters, a carriage return and a line feed. On Unix,
         * the ASCII line feed is used for both purposes - so we can not
         * just use \n, because it would not have a carriage return and the
         * next line will start at the column right after the line feed.
         *
         * This is why text files are different between Unix and MS Windows.
         * In CP/M and derivatives, like MS-DOS and MS Windows, the ASCII
         * standard was strictly adhered to, and therefore a newline requires
         * both a LF and a CR.
         */
        (ttyops->write)(my_tty, "\015\012", 2);
    }
}

static int __init print_string_init(void)
{
    print_string("The module has been inserted.  Hello world!");
    return 0;
}

static void __exit print_string_exit(void)
{
    print_string("The module has been removed.  Farewell world!");
}

module_init(print_string_init);
module_exit(print_string_exit);

MODULE_LICENSE("GPL");
```

13.2 Flashing keyboard LEDs

In certain conditions, you may desire a simpler and more direct way to communicate to the external world. Flashing keyboard LEDs can be such a solution: It is an immediate way to attract attention or to display a status condition. Keyboard LEDs are present on every hardware, they are always visible, they do not need any setup, and their use is rather simple and non-intrusive, compared to writing to a tty or a file.

From v4.14 to v4.15, the timer API made a series of changes to improve memory safety. A buffer overflow in the area of a `timer_list` structure may be able to overwrite the `function` and `data` fields, providing the attacker with a way to use return-oriented programming (ROP) to call arbitrary functions within the kernel. Also, the function prototype of the callback, containing a `unsigned long` argument, will prevent work from any type checking. Further-more, the function prototype with `unsigned long` argument may be an obstacle to the forward-edge protection of *control-flow integrity*. Thus, it is better to use a unique prototype to separate from the cluster that takes an `unsigned long` argument. The timer callback should be passed a pointer to the `timer_list` structure rather than an `unsigned long` argument. Then, it wraps all the information the callback needs, including the `timer_list` structure, into a larger structure, and it can use the `container_of` macro instead of the `unsigned long` value. For more information see: Improving the kernel timers API.

Before Linux v4.14, `setup_timer` was used to initialize the timer and the `timer_list` structure looked like:

```
struct timer_list {
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
    u32 flags;
    /* ... */
};

void setup_timer(struct timer_list *timer, void (*callback)(unsigned long),
                 unsigned long data);
```

Since Linux v4.14, `timer_setup` is adopted and the kernel step by step converting to `timer_setup` from `setup_timer`. One of the reasons why API was changed is it need to coexist with the old version interface. Moreover, the `timer_setup` was implemented by `setup_timer` at first.

```
void timer_setup(struct timer_list *timer,
                 void (*callback)(struct timer_list *), unsigned int flags);
```

The `setup_timer` was then removed since v4.15. As a result, the `timer_list` structure had changed to the following.

```
struct timer_list {
    unsigned long expires;
    void (*function)(struct timer_list *);
    u32 flags;
    /* ... */
};
```

The following source code illustrates a minimal kernel module which, when loaded, starts blinking the keyboard LEDs until it is unloaded.

```
/*
 * kbleds.c - Blink keyboard leds until the module is unloaded.
 */

#include <linux/init.h>
#include <linux/kd.h> /* For KDSETLED */
```

```c
#include <linux/module.h>
#include <linux/tty.h> /* For tty_struct */
#include <linux/vt.h> /* For MAX_NR_CONSOLES */
#include <linux/vt_kern.h> /* for fg_console */
#include <linux/console_struct.h> /* For vc_cons */

MODULE_DESCRIPTION("Example module illustrating the use of Keyboard LEDs.");

static struct timer_list my_timer;
static struct tty_driver *my_driver;
static unsigned long kbledstatus = 0;

#define BLINK_DELAY HZ / 5
#define ALL_LEDS_ON 0x07
#define RESTORE_LEDS 0xFF

/* Function my_timer_func blinks the keyboard LEDs periodically by invoking
 * command KDSETLED of ioctl() on the keyboard driver. To learn more on virtual
 * terminal ioctl operations, please see file:
 *   drivers/tty/vt/vt_ioctl.c, function vt_ioctl().
 *
 * The argument to KDSETLED is alternatively set to 7 (thus causing the led
 * mode to be set to LED_SHOW_IOCTL, and all the leds are lit) and to 0xFF
 * (any value above 7 switches back the led mode to LED_SHOW_FLAGS, thus
 * the LEDs reflect the actual keyboard status).  To learn more on this,
 * please see file: drivers/tty/vt/keyboard.c, function setledstate().
 */
static void my_timer_func(struct timer_list *unused)
{
    struct tty_struct *t = vc_cons[fg_console].d->port.tty;

    if (kbledstatus == ALL_LEDS_ON)
        kbledstatus = RESTORE_LEDS;
    else
        kbledstatus = ALL_LEDS_ON;

    (my_driver->ops->ioctl)(t, KDSETLED, kbledstatus);

    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);
}

static int __init kbleds_init(void)
{
    int i;

    pr_info("kbleds: loading\n");
    pr_info("kbleds: fgconsole is %x\n", fg_console);
    for (i = 0; i < MAX_NR_CONSOLES; i++) {
        if (!vc_cons[i].d)
            break;
        pr_info("poet_atkm: console[%i/%i] #%i, tty %p\n", i, MAX_NR_CONSOLES,
                vc_cons[i].d->vc_num, (void *)vc_cons[i].d->port.tty);
    }
    pr_info("kbleds: finished scanning consoles\n");

```

```
    my_driver = vc_cons[fg_console].d->port.tty->driver;
    pr_info("kbleds: tty driver name %s\n", my_driver->driver_name);

    /* Set up the LED blink timer the first time. */
    timer_setup(&my_timer, my_timer_func, 0);
    my_timer.expires = jiffies + BLINK_DELAY;
    add_timer(&my_timer);

    return 0;
}

static void __exit kbleds_cleanup(void)
{
    pr_info("kbleds: unloading...\n");
    del_timer(&my_timer);
    (my_driver->ops->ioctl)(vc_cons[fg_console].d->port.tty, KDSETLED,
                            RESTORE_LEDS);
}

module_init(kbleds_init);
module_exit(kbleds_cleanup);

MODULE_LICENSE("GPL");
```

If none of the examples in this chapter fit your debugging needs, there might yet be some other tricks to try. Ever wondered what `CONFIG_LL_DEBUG` in `make menuconfig` is good for? If you activate that you get low level access to the serial port. While this might not sound very powerful by itself, you can patch kernel/printk.c or any other essential syscall to print ASCII characters, thus making it possible to trace virtually everything what your code does over a serial line. If you find yourself porting the kernel to some new and former unsupported architecture, this is usually amongst the first things that should be implemented. Logging over a netconsole might also be worth a try.

While you have seen lots of stuff that can be used to aid debugging here, there are some things to be aware of. Debugging is almost always intrusive. Adding debug code can change the situation enough to make the bug seem to disappear. Thus, you should keep debug code to a minimum and make sure it does not show up in production code.

14 Scheduling Tasks

There are two main ways of running tasks: tasklets and work queues. Tasklets are a quick and easy way of scheduling a single function to be run. For example, when triggered from an interrupt, whereas work queues are more complicated but also better suited to running multiple things in a sequence.

It is possible that in future tasklets may be replaced by *threaded irqs*. However, discussion about that has been ongoing since 2007 (Eliminating tasklets), so do not hold your breath. See the section 15.1 if you wish to avoid the tasklet debate.

14.1 Tasklets

Here is an example tasklet module. The `tasklet_fn` function runs for a few seconds. In the meantime, execution of the `example_tasklet_init` function may continue to the exit point, depending on whether it is interrupted by **softirq**.

```
/*
 * example_tasklet.c
 */
#include <linux/delay.h>
#include <linux/interrupt.h>
#include <linux/module.h>
#include <linux/printk.h>

/* Macro DECLARE_TASKLET_OLD exists for compatibility.
 * See https://lwn.net/Articles/830964/
 */
#ifndef DECLARE_TASKLET_OLD
#define DECLARE_TASKLET_OLD(arg1, arg2) DECLARE_TASKLET(arg1, arg2, 0L)
#endif

static void tasklet_fn(unsigned long data)
{
    pr_info("Example tasklet starts\n");
    mdelay(5000);
    pr_info("Example tasklet ends\n");
}

static DECLARE_TASKLET_OLD(mytask, tasklet_fn);

static int __init example_tasklet_init(void)
{
    pr_info("tasklet example init\n");
    tasklet_schedule(&mytask);
    mdelay(200);
    pr_info("Example tasklet init continues...\n");
    return 0;
}

static void __exit example_tasklet_exit(void)
{
    pr_info("tasklet example exit\n");
    tasklet_kill(&mytask);
}

module_init(example_tasklet_init);
module_exit(example_tasklet_exit);

MODULE_DESCRIPTION("Tasklet example");
MODULE_LICENSE("GPL");
```

So with this example loaded `dmesg` should show:

```
tasklet example init
Example tasklet starts
```

```
Example tasklet init continues...
Example tasklet ends
```

Although tasklet is easy to use, it comes with several drawbacks, and developers are discussing about getting rid of tasklet in linux kernel. The tasklet callback runs in atomic context, inside a software interrupt, meaning that it cannot sleep or access user-space data, so not all work can be done in a tasklet handler. Also, the kernel only allows one instance of any given tasklet to be running at any given time; multiple different tasklet callbacks can run in parallel.

In recent kernels, tasklets can be replaced by workqueues, timers, or threaded interrupts.[1] While the removal of tasklets remains a longer-term goal, the current kernel contains more than a hundred uses of tasklets. Now developers are proceeding with the API changes and the macro `DECLARE_TASKLET_OLD` exists for compatibility. For further information, see https://lwn.net/Articles/830964/.

14.2 Work queues

To add a task to the scheduler we can use a workqueue. The kernel then uses the Completely Fair Scheduler (CFS) to execute work within the queue.

```c
/*
 * sched.c
 */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/workqueue.h>

static struct workqueue_struct *queue = NULL;
static struct work_struct work;

static void work_handler(struct work_struct *data)
{
    pr_info("work handler function.\n");
}

static int __init sched_init(void)
{
    queue = alloc_workqueue("HELLOWORLD", WQ_UNBOUND, 1);
    INIT_WORK(&work, work_handler);
    queue_work(queue, &work);
    return 0;
}

static void __exit sched_exit(void)
{
    destroy_workqueue(queue);
}
```

[1] The goal of threaded interrupts is to push more of the work to separate threads, so that the minimum needed for acknowledging an interrupt is reduced, and therefore the time spent handling the interrupt (where it can't handle any other interrupts at the same time) is reduced. See https://lwn.net/Articles/302043/.

```

module_init(sched_init);
module_exit(sched_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Workqueue example");
```

15 Interrupt Handlers

15.1 Interrupt Handlers

Except for the last chapter, everything we did in the kernel so far we have done as a response to a process asking for it, either by dealing with a special file, sending an `ioctl()`, or issuing a system call. But the job of the kernel is not just to respond to process requests. Another job, which is every bit as important, is to speak to the hardware connected to the machine.

There are two types of interaction between the CPU and the rest of the computer's hardware. The first type is when the CPU gives orders to the hardware, the other is when the hardware needs to tell the CPU something. The second, called interrupts, is much harder to implement because it has to be dealt with when convenient for the hardware, not the CPU. Hardware devices typically have a very small amount of RAM, and if you do not read their information when available, it is lost.

Under Linux, hardware interrupts are called IRQ's (Interrupt ReQuests). There are two types of IRQ's, short and long. A short IRQ is one which is expected to take a very short period of time, during which the rest of the machine will be blocked and no other interrupts will be handled. A long IRQ is one which can take longer, and during which other interrupts may occur (but not interrupts from the same device). If at all possible, it is better to declare an interrupt handler to be long.

When the CPU receives an interrupt, it stops whatever it is doing (unless it is processing a more important interrupt, in which case it will deal with this one only when the more important one is done), saves certain parameters on the stack and calls the interrupt handler. This means that certain things are not allowed in the interrupt handler itself, because the system is in an unknown state. Linux kernel solves the problem by splitting interrupt handling into two parts. The first part executes right away and masks the interrupt line. Hardware interrupts must be handled quickly, and that is why we need the second part to handle the heavy work deferred from an interrupt handler. Historically, BH (Linux naming for *Bottom Halves*) statistically book-keeps the deferred functions. **Softirq** and its higher level abstraction, **Tasklet**, replace BH since Linux 2.3.

The way to implement this is to call `request_irq()` to get your interrupt handler called when the relevant IRQ is received.

In practice IRQ handling can be a bit more complex. Hardware is often designed in a way that chains two interrupt controllers, so that all the IRQs from interrupt controller B are cascaded to a certain IRQ from interrupt controller A. Of course, that requires that the kernel finds out which IRQ it really was afterwards and that adds overhead. Other architectures offer some special, very low overhead, so called "fast IRQ" or FIQs. To take advantage of them requires handlers to be written in assembly language, so they do not really fit into the kernel. They can be made to work similar to the others, but after that procedure, they are no longer any faster than "common" IRQs. SMP enabled kernels running on systems with more than one processor need to solve another truckload of problems. It is not enough to know if a certain IRQs has happened, it's also important to know what CPU(s) it was for. People still interested in more details, might want to refer to "APIC" now.

This function receives the IRQ number, the name of the function, flags, a name for `/proc/interrupts` and a parameter to be passed to the interrupt handler. Usually there is a certain number of IRQs available. How many IRQs there are is hardware-dependent.

The flags can be used for specify behaviors of the IRQ. For example, use `IRQF_SHARED` to indicate you are willing to share the IRQ with other interrupt handlers (usually because a number of hardware devices sit on the same IRQ); use the `IRQF_ONESHOT` to indicate that the IRQ is not reenabled after the handler finished. It should be noted that in some materials, you may encouter another set of IRQ flags named with the `SA` prefix. For example, the `SA_SHIRQ` and the `SA_INTERRUPT`. Those are the the IRQ flags in the older kernels. They have been removed completely. Today only the `IRQF` flags are in use. This function will only succeed if there is not already a handler on this IRQ, or if you are both willing to share.

15.2 Detecting button presses

Many popular single board computers, such as Raspberry Pi or Beagleboards, have a bunch of GPIO pins. Attaching buttons to those and then having a button press do something is a classic case in which you might need to use interrupts, so that instead of having the CPU waste time and battery power polling for a change in input state, it is better for the input to trigger the CPU to then run a particular handling function.

Here is an example where buttons are connected to GPIO numbers 17 and 18 and an LED is connected to GPIO 4. You can change those numbers to whatever is appropriate for your board.

```
/*
 * intrpt.c - Handling GPIO with interrupts
 *
 * Based upon the RPi example by Stefan Wendler (devnull@kaltpost.de)
 * from:
 *   https://github.com/wendlers/rpi-kmod-samples
 *
 * Press one button to turn on a LED and another to turn it off.
 */

```

```c
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h> /* for ARRAY_SIZE() */
#include <linux/module.h>
#include <linux/printk.h>

static int button_irqs[] = { -1, -1 };

/* Define GPIOs for LEDs.
 * TODO: Change the numbers for the GPIO on your board.
 */
static struct gpio leds[] = { { 4, GPIOF_OUT_INIT_LOW, "LED 1" } };

/* Define GPIOs for BUTTONS
 * TODO: Change the numbers for the GPIO on your board.
 */
static struct gpio buttons[] = { { 17, GPIOF_IN, "LED 1 ON BUTTON" },
                                 { 18, GPIOF_IN, "LED 1 OFF BUTTON" } };

/* interrupt function triggered when a button is pressed. */
static irqreturn_t button_isr(int irq, void *data)
{
    /* first button */
    if (irq == button_irqs[0] && !gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 1);
    /* second button */
    else if (irq == button_irqs[1] && gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 0);

    return IRQ_HANDLED;
}

static int __init intrpt_init(void)
{
    int ret = 0;

    pr_info("%s\n", __func__);

    /* register LED gpios */
    ret = gpio_request_array(leds, ARRAY_SIZE(leds));

    if (ret) {
        pr_err("Unable to request GPIOs for LEDs: %d\n", ret);
        return ret;
    }

    /* register BUTTON gpios */
    ret = gpio_request_array(buttons, ARRAY_SIZE(buttons));

    if (ret) {
        pr_err("Unable to request GPIOs for BUTTONs: %d\n", ret);
        goto fail1;
    }

    pr_info("Current button1 value: %d\n", gpio_get_value(buttons[0].gpio));

    ret = gpio_to_irq(buttons[0].gpio);
```

```

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[0] = ret;

    pr_info("Successfully requested BUTTON1 IRQ # %d\n", button_irqs[0]);

    ret = request_irq(button_irqs[0], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button1", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    ret = gpio_to_irq(buttons[1].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[1] = ret;

    pr_info("Successfully requested BUTTON2 IRQ # %d\n", button_irqs[1]);

    ret = request_irq(button_irqs[1], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button2", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail3;
    }

    return 0;

/* cleanup what has been setup so far */
fail3:
    free_irq(button_irqs[0], NULL);

fail2:
    gpio_free_array(buttons, ARRAY_SIZE(leds));

fail1:
    gpio_free_array(leds, ARRAY_SIZE(leds));

    return ret;
}

static void __exit intrpt_exit(void)
{
    int i;
```

```

    pr_info("%s\n", __func__);

    /* free irqs */
    free_irq(button_irqs[0], NULL);
    free_irq(button_irqs[1], NULL);

    /* turn all LEDs off */
    for (i = 0; i < ARRAY_SIZE(leds); i++)
        gpio_set_value(leds[i].gpio, 0);

    /* unregister */
    gpio_free_array(leds, ARRAY_SIZE(leds));
    gpio_free_array(buttons, ARRAY_SIZE(buttons));
}

module_init(intrpt_init);
module_exit(intrpt_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Handle some GPIO interrupts");
```

15.3 Bottom Half

Suppose you want to do a bunch of stuff inside of an interrupt routine. A common way to do that without rendering the interrupt unavailable for a significant duration is to combine it with a tasklet. This pushes the bulk of the work off into the scheduler.

The example below modifies the previous example to also run an additional task when an interrupt is triggered.

```
/*
 * bottomhalf.c - Top and bottom half interrupt handling
 *
 * Based upon the RPi example by Stefan Wendler (devnull@kaltpost.de)
 * from:
 *    https://github.com/wendlers/rpi-kmod-samples
 *
 * Press one button to turn on an LED and another to turn it off
 */

#include <linux/delay.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/module.h>
#include <linux/printk.h>
#include <linux/init.h>

/* Macro DECLARE_TASKLET_OLD exists for compatibility.
 * See https://lwn.net/Articles/830964/
 */
#ifndef DECLARE_TASKLET_OLD
#define DECLARE_TASKLET_OLD(arg1, arg2) DECLARE_TASKLET(arg1, arg2, 0L)
#endif
```

```c

static int button_irqs[] = { -1, -1 };

/* Define GPIOs for LEDs.
 * TODO: Change the numbers for the GPIO on your board.
 */
static struct gpio leds[] = { { 4, GPIOF_OUT_INIT_LOW, "LED 1" } };

/* Define GPIOs for BUTTONS
 * TODO: Change the numbers for the GPIO on your board.
 */
static struct gpio buttons[] = {
    { 17, GPIOF_IN, "LED 1 ON BUTTON" },
    { 18, GPIOF_IN, "LED 1 OFF BUTTON" },
};

/* Tasklet containing some non-trivial amount of processing */
static void bottomhalf_tasklet_fn(unsigned long data)
{
    pr_info("Bottom half tasklet starts\n");
    /* do something which takes a while */
    mdelay(500);
    pr_info("Bottom half tasklet ends\n");
}

static DECLARE_TASKLET_OLD(buttontask, bottomhalf_tasklet_fn);

/* interrupt function triggered when a button is pressed */
static irqreturn_t button_isr(int irq, void *data)
{
    /* Do something quickly right now */
    if (irq == button_irqs[0] && !gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 1);
    else if (irq == button_irqs[1] && gpio_get_value(leds[0].gpio))
        gpio_set_value(leds[0].gpio, 0);

    /* Do the rest at leisure via the scheduler */
    tasklet_schedule(&buttontask);

    return IRQ_HANDLED;
}

static int __init bottomhalf_init(void)
{
    int ret = 0;

    pr_info("%s\n", __func__);

    /* register LED gpios */
    ret = gpio_request_array(leds, ARRAY_SIZE(leds));

    if (ret) {
        pr_err("Unable to request GPIOs for LEDs: %d\n", ret);
        return ret;
    }

    /* register BUTTON gpios */
```

```
    ret = gpio_request_array(buttons, ARRAY_SIZE(buttons));

    if (ret) {
        pr_err("Unable to request GPIOs for BUTTONs: %d\n", ret);
        goto fail1;
    }

    pr_info("Current button1 value: %d\n", gpio_get_value(buttons[0].gpio));

    ret = gpio_to_irq(buttons[0].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[0] = ret;

    pr_info("Successfully requested BUTTON1 IRQ # %d\n", button_irqs[0]);

    ret = request_irq(button_irqs[0], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button1", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    ret = gpio_to_irq(buttons[1].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[1] = ret;

    pr_info("Successfully requested BUTTON2 IRQ # %d\n", button_irqs[1]);

    ret = request_irq(button_irqs[1], button_isr,
                      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                      "gpiomod#button2", NULL);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail3;
    }

    return 0;

/* cleanup what has been setup so far */
fail3:
    free_irq(button_irqs[0], NULL);

fail2:
    gpio_free_array(buttons, ARRAY_SIZE(leds));
```

```

fail1:
    gpio_free_array(leds, ARRAY_SIZE(leds));

    return ret;
}

static void __exit bottomhalf_exit(void)
{
    int i;

    pr_info("%s\n", __func__);

    /* free irqs */
    free_irq(button_irqs[0], NULL);
    free_irq(button_irqs[1], NULL);

    /* turn all LEDs off */
    for (i = 0; i < ARRAY_SIZE(leds); i++)
        gpio_set_value(leds[i].gpio, 0);

    /* unregister */
    gpio_free_array(leds, ARRAY_SIZE(leds));
    gpio_free_array(buttons, ARRAY_SIZE(buttons));
}

module_init(bottomhalf_init);
module_exit(bottomhalf_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Interrupt with top and bottom half");
```

15.4 Threaded IRQ

Threaded IRQ is a mechanism to organize both top-half and bottom-half of an IRQ at once. A threaded IRQ splits the one handler in `request_irq()` into two: one for the top-half, the other for the bottom-half. The `request_threaded_irq()` is the function for using threaded IRQs. Two handlers are registered at once in the `request_threaded_irq()`.

Those two handlers run in different context. The top-half handler runs in interrupt context. It's the equivalence of the handler passed to the `request_irq()`. The bottom-half handler on the other hand runs in its own thread. This thread is created on registration of a threaded IRQ. Its sole purpose is to run this bottom-half handler. This is where a threaded IRQ is "threaded". If `IRQ_WAKE_THREAD` is returned by the top-half handler, that bottom-half serving thread will wake up. The thread then runs the bottom-half handler.

Here is an example of how to do the same thing as before, with top and bottom halves, but using threads.

```
/*
 * bh_thread.c - Top and bottom half interrupt handling
 *
```

```c
 * Based upon the RPi example by Stefan Wendler (devnull@kaltpost.de)
 * from:
 *    https://github.com/wendlers/rpi-kmod-samples
 *
 * Press one button to turn on a LED and another to turn it off
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/delay.h>
#include <linux/interrupt.h>

static int button_irqs[] = { -1, -1 };

/* Define GPIOs for LEDs.
 * FIXME: Change the numbers for the GPIO on your board.
 */
static struct gpio leds[] = { { 4, GPIOF_OUT_INIT_LOW, "LED 1" } };

/* Define GPIOs for BUTTONS
 * FIXME: Change the numbers for the GPIO on your board.
 */
static struct gpio buttons[] = {
    { 17, GPIOF_IN, "LED 1 ON BUTTON" },
    { 18, GPIOF_IN, "LED 1 OFF BUTTON" },
};

/* This happens immediately, when the IRQ is triggered */
static irqreturn_t button_top_half(int irq, void *ident)
{
    return IRQ_WAKE_THREAD;
}

/* This can happen at leisure, freeing up IRQs for other high priority task */
static irqreturn_t button_bottom_half(int irq, void *ident)
{
    pr_info("Bottom half task starts\n");
    mdelay(500); /* do something which takes a while */
    pr_info("Bottom half task ends\n");
    return IRQ_HANDLED;
}

static int __init bottomhalf_init(void)
{
    int ret = 0;

    pr_info("%s\n", __func__);

    /* register LED gpios */
    ret = gpio_request_array(leds, ARRAY_SIZE(leds));

    if (ret) {
        pr_err("Unable to request GPIOs for LEDs: %d\n", ret);
        return ret;
    }

```

```c
    /* register BUTTON gpios */
    ret = gpio_request_array(buttons, ARRAY_SIZE(buttons));

    if (ret) {
        pr_err("Unable to request GPIOs for BUTTONs: %d\n", ret);
        goto fail1;
    }

    pr_info("Current button1 value: %d\n", gpio_get_value(buttons[0].gpio));

    ret = gpio_to_irq(buttons[0].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[0] = ret;

    pr_info("Successfully requested BUTTON1 IRQ # %d\n", button_irqs[0]);

    ret = request_threaded_irq(button_irqs[0], button_top_half,
                               button_bottom_half,
                               IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                               "gpiomod#button1", &buttons[0]);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    ret = gpio_to_irq(buttons[1].gpio);

    if (ret < 0) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail2;
    }

    button_irqs[1] = ret;

    pr_info("Successfully requested BUTTON2 IRQ # %d\n", button_irqs[1]);

    ret = request_threaded_irq(button_irqs[1], button_top_half,
                               button_bottom_half,
                               IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                               "gpiomod#button2", &buttons[1]);

    if (ret) {
        pr_err("Unable to request IRQ: %d\n", ret);
        goto fail3;
    }

    return 0;

/* cleanup what has been setup so far */
fail3:
    free_irq(button_irqs[0], NULL);
```

```c

fail2:
    gpio_free_array(buttons, ARRAY_SIZE(leds));

fail1:
    gpio_free_array(leds, ARRAY_SIZE(leds));

    return ret;
}

static void __exit bottomhalf_exit(void)
{
    int i;

    pr_info("%s\n", __func__);

    /* free irqs */
    free_irq(button_irqs[0], NULL);
    free_irq(button_irqs[1], NULL);

    /* turn all LEDs off */
    for (i = 0; i < ARRAY_SIZE(leds); i++)
        gpio_set_value(leds[i].gpio, 0);

    /* unregister */
    gpio_free_array(leds, ARRAY_SIZE(leds));
    gpio_free_array(buttons, ARRAY_SIZE(buttons));
}

module_init(bottomhalf_init);
module_exit(bottomhalf_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Interrupt with top and bottom half");
```

A threaded IRQ is registered using `request_threaded_irq()`. This function only takes one additional parameter than the `request_irq()` – the bottom-half handling function that runs in its own thread. In this example it is the `button_bottom_half()`. Usage of other parameters are the same as `request_irq()`.

Presence of both handlers is not mandatory. If either of them is not needed, pass the `NULL` instead. A `NULL` top-half handler implies that no action is taken except to wake up the bottom-half serving thread, which runs the bottom-half handler. Similarly, a `NULL` bottom-half handler effectively acts as if `request_irq()` were used. In fact, this is how `request_irq()` is implemented.

Note that passing `NULL` to both handlers is considered an error and will make registration fail.

16 Virtual Input Device Driver

The input device driver is a module that provides a way to communicate with the interaction device via the event. For example, the keyboard can send the press or release event to tell the kernel what we want to do. The input device driver will allocate a new input structure with `input_allocate_device()` and sets up input bitfields, device id, version, etc. After that, registers it by calling `input_register_device()`.

Here is an example, vinput, It is an API to allow easy development of virtual input drivers. The drivers needs to export a `vinput_device()` that contains the virtual device name and `vinput_ops` structure that describes:

- the init function: `init()`
- the input event injection function: `send()`
- the readback function: `read()`

Then using `vinput_register_device()` and `vinput_unregister_device()` will add a new device to the list of support virtual input devices.

```
int init(struct vinput *);
```

This function is passed a `struct vinput` already initialized with an allocated `struct input_dev`. The `init()` function is responsible for initializing the capabilities of the input device and register it.

```
int send(struct vinput *, char *, int);
```

This function will receive a user string to interpret and inject the event using the `input_report_XXXX` or `input_event` call. The string is already copied from user.

```
int read(struct vinput *, char *, int);
```

This function is used for debugging and should fill the buffer parameter with the last event sent in the virtual input device format. The buffer will then be copied to user.

vinput devices are created and destroyed using sysfs. And, event injection is done through a `/dev` node. The device name will be used by the userland to export a new virtual input device.

The `class_attribute` structure is similar to other attribute types we talked about in section 8:

```
struct class_attribute {
    struct attribute attr;
    ssize_t (*show)(struct class *class, struct class_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct class *class, struct class_attribute *attr,
                    const char *buf, size_t count);
};
```

In `vinput.c`, the macro `CLASS_ATTR_WO(export/unexport)` defined in include/linux/device.h (in this case, `device.h` is included in include/linux/input.h) will generate the `class_attribute` structures which are named `class_attr_export/unexport`. Then, put them into `vinput_class_attrs` array and the macro `ATTRIBUTE_GROUPS(vinput_class)` will generate the `struct attribute_group vinput_class_group` that should be assigned in `vinput_class`. Finally, call `class_register(&vinput_class)` to create attributes in sysfs.

To create a `vinputX` sysfs entry and `/dev` node.

```
echo "vkbd" | sudo tee /sys/class/vinput/export
```

To unexport the device, just echo its id in unexport:

```
echo "0" | sudo tee /sys/class/vinput/unexport
```

```
/*
 * vinput.h
 */

#ifndef VINPUT_H
#define VINPUT_H

#include <linux/input.h>
#include <linux/spinlock.h>

#define VINPUT_MAX_LEN 128
#define MAX_VINPUT 32
#define VINPUT_MINORS MAX_VINPUT

#define dev_to_vinput(dev) container_of(dev, struct vinput, dev)

struct vinput_device;

struct vinput {
    long id;
    long devno;
    long last_entry;
    spinlock_t lock;

    void *priv_data;

    struct device dev;
    struct list_head list;
    struct input_dev *input;
    struct vinput_device *type;
};

struct vinput_ops {
    int (*init)(struct vinput *);
    int (*kill)(struct vinput *);
    int (*send)(struct vinput *, char *, int);
```

```
    int (*read)(struct vinput *, char *, int);
};

struct vinput_device {
    char name[16];
    struct list_head list;
    struct vinput_ops *ops;
};

int vinput_register(struct vinput_device *dev);
void vinput_unregister(struct vinput_device *dev);

#endif
```

```
/*
 * vinput.c
 */

#include <linux/cdev.h>
#include <linux/input.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/spinlock.h>
#include <linux/version.h>

#include <asm/uaccess.h>

#include "vinput.h"

#define DRIVER_NAME "vinput"

#define dev_to_vinput(dev) container_of(dev, struct vinput, dev)

static DECLARE_BITMAP(vinput_ids, VINPUT_MINORS);

static LIST_HEAD(vinput_devices);
static LIST_HEAD(vinput_vdevices);

static int vinput_dev;
static struct spinlock vinput_lock;
static struct class vinput_class;

/* Search the name of vinput device in the vinput_devices linked list,
 * which added at vinput_register().
 */
static struct vinput_device *vinput_get_device_by_type(const char *type)
{
    int found = 0;
    struct vinput_device *vinput;
    struct list_head *curr;

    spin_lock(&vinput_lock);
    list_for_each (curr, &vinput_devices) {
        vinput = list_entry(curr, struct vinput_device, list);
        if (vinput && strncmp(type, vinput->name, strlen(vinput->name)) == 0)
        ↪  {
            found = 1;
```

```c
            break;
        }
    }
    spin_unlock(&vinput_lock);

    if (found)
        return vinput;
    return ERR_PTR(-ENODEV);
}

/* Search the id of virtual device in the vinput_vdevices linked list,
 * which added at vinput_alloc_vdevice().
 */
static struct vinput *vinput_get_vdevice_by_id(long id)
{
    struct vinput *vinput = NULL;
    struct list_head *curr;

    spin_lock(&vinput_lock);
    list_for_each (curr, &vinput_vdevices) {
        vinput = list_entry(curr, struct vinput, list);
        if (vinput && vinput->id == id)
            break;
    }
    spin_unlock(&vinput_lock);

    if (vinput && vinput->id == id)
        return vinput;
    return ERR_PTR(-ENODEV);
}

static int vinput_open(struct inode *inode, struct file *file)
{
    int err = 0;
    struct vinput *vinput = NULL;

    vinput = vinput_get_vdevice_by_id(iminor(inode));

    if (IS_ERR(vinput))
        err = PTR_ERR(vinput);
    else
        file->private_data = vinput;

    return err;
}

static int vinput_release(struct inode *inode, struct file *file)
{
    return 0;
}

static ssize_t vinput_read(struct file *file, char __user *buffer, size_t
↪  count,
                           loff_t *offset)
{
    int len;
    char buff[VINPUT_MAX_LEN + 1];
```

```
    struct vinput *vinput = file->private_data;

    len = vinput->type->ops->read(vinput, buff, count);

    if (*offset > len)
        count = 0;
    else if (count + *offset > VINPUT_MAX_LEN)
        count = len - *offset;

    if (raw_copy_to_user(buffer, buff + *offset, count))
        return -EFAULT;

    *offset += count;

    return count;
}

static ssize_t vinput_write(struct file *file, const char __user *buffer,
                            size_t count, loff_t *offset)
{
    char buff[VINPUT_MAX_LEN + 1];
    struct vinput *vinput = file->private_data;

    memset(buff, 0, sizeof(char) * (VINPUT_MAX_LEN + 1));

    if (count > VINPUT_MAX_LEN) {
        dev_warn(&vinput->dev, "Too long. %d bytes allowed\n",
            VINPUT_MAX_LEN);
        return -EINVAL;
    }

    if (raw_copy_from_user(buff, buffer, count))
        return -EFAULT;

    return vinput->type->ops->send(vinput, buff, count);
}

static const struct file_operations vinput_fops = {
#if LINUX_VERSION_CODE < KERNEL_VERSION(6, 4, 0)
    .owner = THIS_MODULE,
#endif
    .open = vinput_open,
    .release = vinput_release,
    .read = vinput_read,
    .write = vinput_write,
};

static void vinput_unregister_vdevice(struct vinput *vinput)
{
    input_unregister_device(vinput->input);
    if (vinput->type->ops->kill)
        vinput->type->ops->kill(vinput);
}

static void vinput_destroy_vdevice(struct vinput *vinput)
{
    /* Remove from the list first */
```

```c
    spin_lock(&vinput_lock);
    list_del(&vinput->list);
    clear_bit(vinput->id, vinput_ids);
    spin_unlock(&vinput_lock);

    module_put(THIS_MODULE);

    kfree(vinput);
}

static void vinput_release_dev(struct device *dev)
{
    struct vinput *vinput = dev_to_vinput(dev);
    int id = vinput->id;

    vinput_destroy_vdevice(vinput);

    pr_debug("released vinput%d.\n", id);
}

static struct vinput *vinput_alloc_vdevice(void)
{
    int err;
    struct vinput *vinput = kzalloc(sizeof(struct vinput), GFP_KERNEL);

    if (!vinput) {
        pr_err("vinput: Cannot allocate vinput input device\n");
        return ERR_PTR(-ENOMEM);
    }

    try_module_get(THIS_MODULE);

    spin_lock_init(&vinput->lock);

    spin_lock(&vinput_lock);
    vinput->id = find_first_zero_bit(vinput_ids, VINPUT_MINORS);
    if (vinput->id >= VINPUT_MINORS) {
        err = -ENOBUFS;
        goto fail_id;
    }
    set_bit(vinput->id, vinput_ids);
    list_add(&vinput->list, &vinput_vdevices);
    spin_unlock(&vinput_lock);

    /* allocate the input device */
    vinput->input = input_allocate_device();
    if (vinput->input == NULL) {
        pr_err("vinput: Cannot allocate vinput input device\n");
        err = -ENOMEM;
        goto fail_input_dev;
    }

    /* initialize device */
    vinput->dev.class = &vinput_class;
    vinput->dev.release = vinput_release_dev;
    vinput->dev.devt = MKDEV(vinput_dev, vinput->id);
    dev_set_name(&vinput->dev, DRIVER_NAME "%lu", vinput->id);
```

```
    return vinput;

fail_input_dev:
    spin_lock(&vinput_lock);
    list_del(&vinput->list);
fail_id:
    spin_unlock(&vinput_lock);
    module_put(THIS_MODULE);
    kfree(vinput);

    return ERR_PTR(err);
}

static int vinput_register_vdevice(struct vinput *vinput)
{
    int err = 0;

    /* register the input device */
    vinput->input->name = vinput->type->name;
    vinput->input->phys = "vinput";
    vinput->input->dev.parent = &vinput->dev;

    vinput->input->id.bustype = BUS_VIRTUAL;
    vinput->input->id.product = 0x0000;
    vinput->input->id.vendor = 0x0000;
    vinput->input->id.version = 0x0000;

    err = vinput->type->ops->init(vinput);

    if (err == 0)
        dev_info(&vinput->dev, "Registered virtual input %s %ld\n",
                 vinput->type->name, vinput->id);

    return err;
}

#if LINUX_VERSION_CODE >= KERNEL_VERSION(6, 4, 0)
static ssize_t export_store(const struct class *class,
                            const struct class_attribute *attr,
#else
static ssize_t export_store(struct class *class, struct class_attribute *attr,
#endif
                            const char *buf, size_t len)
{
    int err;
    struct vinput *vinput;
    struct vinput_device *device;

    device = vinput_get_device_by_type(buf);
    if (IS_ERR(device)) {
        pr_info("vinput: This virtual device isn't registered\n");
        err = PTR_ERR(device);
        goto fail;
    }

    vinput = vinput_alloc_vdevice();
```

```c
    if (IS_ERR(vinput)) {
        err = PTR_ERR(vinput);
        goto fail;
    }

    vinput->type = device;
    err = device_register(&vinput->dev);
    if (err < 0)
        goto fail_register;

    err = vinput_register_vdevice(vinput);
    if (err < 0)
        goto fail_register_vinput;

    return len;

fail_register_vinput:
    device_unregister(&vinput->dev);
fail_register:
    vinput_destroy_vdevice(vinput);
fail:
    return err;
}
/* This macro generates class_attr_export structure and export_store() */
static CLASS_ATTR_WO(export);

#if LINUX_VERSION_CODE >= KERNEL_VERSION(6, 4, 0)
static ssize_t unexport_store(const struct class *class,
                              const struct class_attribute *attr,
#else
static ssize_t unexport_store(struct class *class, struct class_attribute
↪  *attr,
#endif
                              const char *buf, size_t len)
{
    int err;
    unsigned long id;
    struct vinput *vinput;

    err = kstrtol(buf, 10, &id);
    if (err) {
        err = -EINVAL;
        goto failed;
    }

    vinput = vinput_get_vdevice_by_id(id);
    if (IS_ERR(vinput)) {
        pr_err("vinput: No such vinput device %ld\n", id);
        err = PTR_ERR(vinput);
        goto failed;
    }

    vinput_unregister_vdevice(vinput);
    device_unregister(&vinput->dev);

    return len;
failed:
```

```
    return err;
}
/* This macro generates class_attr_unexport structure and unexport_store() */
static CLASS_ATTR_WO(unexport);

static struct attribute *vinput_class_attrs[] = {
    &class_attr_export.attr,
    &class_attr_unexport.attr,
    NULL,
};

/* This macro generates vinput_class_groups structure */
ATTRIBUTE_GROUPS(vinput_class);

static struct class vinput_class = {
    .name = "vinput",
#if LINUX_VERSION_CODE < KERNEL_VERSION(6, 4, 0)
    .owner = THIS_MODULE,
#endif
    .class_groups = vinput_class_groups,
};

int vinput_register(struct vinput_device *dev)
{
    spin_lock(&vinput_lock);
    list_add(&dev->list, &vinput_devices);
    spin_unlock(&vinput_lock);

    pr_info("vinput: registered new virtual input device '%s'\n", dev->name);

    return 0;
}
EXPORT_SYMBOL(vinput_register);

void vinput_unregister(struct vinput_device *dev)
{
    struct list_head *curr, *next;

    /* Remove from the list first */
    spin_lock(&vinput_lock);
    list_del(&dev->list);
    spin_unlock(&vinput_lock);

    /* unregister all devices of this type */
    list_for_each_safe (curr, next, &vinput_vdevices) {
        struct vinput *vinput = list_entry(curr, struct vinput, list);
        if (vinput && vinput->type == dev) {
            vinput_unregister_vdevice(vinput);
            device_unregister(&vinput->dev);
        }
    }

    pr_info("vinput: unregistered virtual input device '%s'\n", dev->name);
}
EXPORT_SYMBOL(vinput_unregister);

static int __init vinput_init(void)
```

```
{
    int err = 0;

    pr_info("vinput: Loading virtual input driver\n");

    vinput_dev = register_chrdev(0, DRIVER_NAME, &vinput_fops);
    if (vinput_dev < 0) {
        pr_err("vinput: Unable to allocate char dev region\n");
        err = vinput_dev;
        goto failed_alloc;
    }

    spin_lock_init(&vinput_lock);

    err = class_register(&vinput_class);
    if (err < 0) {
        pr_err("vinput: Unable to register vinput class\n");
        goto failed_class;
    }

    return 0;
failed_class:
    class_unregister(&vinput_class);
failed_alloc:
    return err;
}

static void __exit vinput_end(void)
{
    pr_info("vinput: Unloading virtual input driver\n");

    unregister_chrdev(vinput_dev, DRIVER_NAME);
    class_unregister(&vinput_class);
}

module_init(vinput_init);
module_exit(vinput_end);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Emulate input events");
```

Here the virtual keyboard is one of example to use vinput. It supports all `KEY_MAX` keycodes. The injection format is the `KEY_CODE` such as defined in include/linux/input.h. A positive value means `KEY_PRESS` while a negative value is a `KEY_RELEASE`. The keyboard supports repetition when the key stays pressed for too long. The following demonstrates how simulation work.

Simulate a key press on "g" (`KEY_G` = 34):

```
echo "+34" | sudo tee /dev/vinput0
```

Simulate a key release on "g" (`KEY_G` = 34):

```
echo "-34" | sudo tee /dev/vinput0
```

```
/*
 * vkbd.c
 */

#include <linux/init.h>
#include <linux/input.h>
#include <linux/module.h>
#include <linux/spinlock.h>

#include "vinput.h"

#define VINPUT_KBD "vkbd"
#define VINPUT_RELEASE 0
#define VINPUT_PRESS 1

static unsigned short vkeymap[KEY_MAX];

static int vinput_vkbd_init(struct vinput *vinput)
{
    int i;

    /* Set up the input bitfield */
    vinput->input->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);
    vinput->input->keycodesize = sizeof(unsigned short);
    vinput->input->keycodemax = KEY_MAX;
    vinput->input->keycode = vkeymap;

    for (i = 0; i < KEY_MAX; i++)
        set_bit(vkeymap[i], vinput->input->keybit);

    /* vinput will help us allocate new input device structure via
     * input_allocate_device(). So, we can register it straightforwardly.
     */
    return input_register_device(vinput->input);
}

static int vinput_vkbd_read(struct vinput *vinput, char *buff, int len)
{
    spin_lock(&vinput->lock);
    len = snprintf(buff, len, "%+ld\n", vinput->last_entry);
    spin_unlock(&vinput->lock);

    return len;
}

static int vinput_vkbd_send(struct vinput *vinput, char *buff, int len)
{
    int ret;
    long key = 0;
    short type = VINPUT_PRESS;

    /* Determine which event was received (press or release)
     * and store the state.
```

```
     */
    if (buff[0] == '+')
        ret = kstrtol(buff + 1, 10, &key);
    else
        ret = kstrtol(buff, 10, &key);
    if (ret)
        dev_err(&vinput->dev, "error during kstrtol: -%d\n", ret);
    spin_lock(&vinput->lock);
    vinput->last_entry = key;
    spin_unlock(&vinput->lock);

    if (key < 0) {
        type = VINPUT_RELEASE;
        key = -key;
    }

    dev_info(&vinput->dev, "Event %s code %ld\n",
             (type == VINPUT_RELEASE) ? "VINPUT_RELEASE" : "VINPUT_PRESS",
             ↪  key);

    /* Report the state received to input subsystem. */
    input_report_key(vinput->input, key, type);
    /* Tell input subsystem that it finished the report. */
    input_sync(vinput->input);

    return len;
}

static struct vinput_ops vkbd_ops = {
    .init = vinput_vkbd_init,
    .send = vinput_vkbd_send,
    .read = vinput_vkbd_read,
};

static struct vinput_device vkbd_dev = {
    .name = VINPUT_KBD,
    .ops = &vkbd_ops,
};

static int __init vkbd_init(void)
{
    int i;

    for (i = 0; i < KEY_MAX; i++)
        vkeymap[i] = i;
    return vinput_register(&vkbd_dev);
}

static void __exit vkbd_end(void)
{
    vinput_unregister(&vkbd_dev);
}

module_init(vkbd_init);
module_exit(vkbd_end);

MODULE_LICENSE("GPL");
```

```
MODULE_DESCRIPTION("Emulate keyboard input events through /dev/vinput");
```

17 Standardizing the interfaces: The Device Model

Up to this point we have seen all kinds of modules doing all kinds of things, but there was no consistency in their interfaces with the rest of the kernel. To impose some consistency such that there is at minimum a standardized way to start, suspend and resume a device model was added. An example is shown below, and you can use this as a template to add your own suspend, resume or other interface functions.

```c
/*
 * devicemodel.c
 */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/platform_device.h>

struct devicemodel_data {
    char *greeting;
    int number;
};

static int devicemodel_probe(struct platform_device *dev)
{
    struct devicemodel_data *pd =
        (struct devicemodel_data *)(dev->dev.platform_data);

    pr_info("devicemodel probe\n");
    pr_info("devicemodel greeting: %s; %d\n", pd->greeting, pd->number);

    /* Your device initialization code */

    return 0;
}

static int devicemodel_remove(struct platform_device *dev)
{
    pr_info("devicemodel example removed\n");

    /* Your device removal code */

    return 0;
}

static int devicemodel_suspend(struct device *dev)
{
    pr_info("devicemodel example suspend\n");

    /* Your device suspend code */

    return 0;
}
```

```
static int devicemodel_resume(struct device *dev)
{
    pr_info("devicemodel example resume\n");

    /* Your device resume code */

    return 0;
}

static const struct dev_pm_ops devicemodel_pm_ops = {
    .suspend = devicemodel_suspend,
    .resume = devicemodel_resume,
    .poweroff = devicemodel_suspend,
    .freeze = devicemodel_suspend,
    .thaw = devicemodel_resume,
    .restore = devicemodel_resume,
};

static struct platform_driver devicemodel_driver = {
    .driver =
        {
            .name = "devicemodel_example",
            .pm = &devicemodel_pm_ops,
        },
    .probe = devicemodel_probe,
    .remove = devicemodel_remove,
};

static int __init devicemodel_init(void)
{
    int ret;

    pr_info("devicemodel init\n");

    ret = platform_driver_register(&devicemodel_driver);

    if (ret) {
        pr_err("Unable to register driver\n");
        return ret;
    }

    return 0;
}

static void __exit devicemodel_exit(void)
{
    pr_info("devicemodel exit\n");
    platform_driver_unregister(&devicemodel_driver);
}

module_init(devicemodel_init);
module_exit(devicemodel_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Linux Device Model example");
```

18 Optimizations

18.1 Likely and Unlikely conditions

Sometimes you might want your code to run as quickly as possible, especially if it is handling an interrupt or doing something which might cause noticeable latency. If your code contains boolean conditions and if you know that the conditions are almost always likely to evaluate as either `true` or `false`, then you can allow the compiler to optimize for this using the `likely` and `unlikely` macros. For example, when allocating memory you are almost always expecting this to succeed.

```
bvl = bvec_alloc(gfp_mask, nr_iovecs, &idx);
if (unlikely(!bvl)) {
    mempool_free(bio, bio_pool);
    bio = NULL;
    goto out;
}
```

When the `unlikely` macro is used, the compiler alters its machine instruction output, so that it continues along the false branch and only jumps if the condition is true. That avoids flushing the processor pipeline. The opposite happens if you use the `likely` macro.

18.2 Static keys

Static keys allow us to enable or disable kernel code paths based on the runtime state of key. Its APIs have been available since 2010 (most architectures are already supported), use self-modifying code to eliminate the overhead of cache and branch prediction. The most typical use case of static keys is for performance-sensitive kernel code, such as tracepoints, context switching, networking, etc. These hot paths of the kernel often contain branches and can be optimized easily using this technique. Before we can use static keys in the kernel, we need to make sure that gcc supports `asm goto` inline assembly, and the following kernel configurations are set:

```
CONFIG_JUMP_LABEL=y
CONFIG_HAVE_ARCH_JUMP_LABEL=y
CONFIG_HAVE_ARCH_JUMP_LABEL_RELATIVE=y
```

To declare a static key, we need to define a global variable using the `DEFINE_STATIC_KEY_FALSE` or `DEFINE_STATIC_KEY_TRUE` macro defined in include/linux/jump_label.h. This macro initializes the key with the given initial value, which is either false or true, respectively. For example, to declare a static key with an initial value of false, we can use the following code:

```
DEFINE_STATIC_KEY_FALSE(fkey);
```

Once the static key has been declared, we need to add branching code to the module that uses the static key. For example, the code includes a fastpath, where a no-op instruction will be generated at compile time as the key is initialized to false and the branch is unlikely to be taken.

```
pr_info("fastpath 1\n");
if (static_branch_unlikely(&fkey))
    pr_alert("do unlikely thing\n");
pr_info("fastpath 2\n");
```

If the key is enabled at runtime by calling `static_branch_enable(&fkey)`, the fastpath will be patched with an unconditional jump instruction to the slowpath code `pr_alert`, so the branch will always be taken until the key is disabled again.

The following kernel module derived from `chardev.c`, demonstrates how the static key works.

```
/*
 * static_key.c
 */

#include <linux/atomic.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/kernel.h> /* for sprintf() */
#include <linux/module.h>
#include <linux/printk.h>
#include <linux/types.h>
#include <linux/uaccess.h> /* for get_user and put_user */
#include <linux/jump_label.h> /* for static key macros */
#include <linux/version.h>

#include <asm/errno.h>

static int device_open(struct inode *inode, struct file *file);
static int device_release(struct inode *inode, struct file *file);
static ssize_t device_read(struct file *file, char __user *buf, size_t count,
                           loff_t *ppos);
static ssize_t device_write(struct file *file, const char __user *buf,
                            size_t count, loff_t *ppos);

#define SUCCESS 0
#define DEVICE_NAME "key_state"
#define BUF_LEN 10

static int major;

enum {
    CDEV_NOT_USED = 0,
    CDEV_EXCLUSIVE_OPEN = 1,
};

static atomic_t already_open = ATOMIC_INIT(CDEV_NOT_USED);
```

```c

static char msg[BUF_LEN + 1];

static struct class *cls;

static DEFINE_STATIC_KEY_FALSE(fkey);

static struct file_operations chardev_fops = {
#if LINUX_VERSION_CODE < KERNEL_VERSION(6, 4, 0)
    .owner = THIS_MODULE,
#endif
    .open = device_open,
    .release = device_release,
    .read = device_read,
    .write = device_write,
};

static int __init chardev_init(void)
{
    major = register_chrdev(0, DEVICE_NAME, &chardev_fops);
    if (major < 0) {
        pr_alert("Registering char device failed with %d\n", major);
        return major;
    }

    pr_info("I was assigned major number %d\n", major);

#if LINUX_VERSION_CODE < KERNEL_VERSION(6, 4, 0)
    cls = class_create(THIS_MODULE, DEVICE_NAME);
#else
    cls = class_create(DEVICE_NAME);
#endif

    device_create(cls, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);

    pr_info("Device created on /dev/%s\n", DEVICE_NAME);

    return SUCCESS;
}

static void __exit chardev_exit(void)
{
    device_destroy(cls, MKDEV(major, 0));
    class_destroy(cls);

    /* Unregister the device */
    unregister_chrdev(major, DEVICE_NAME);
}

/* Methods */

/**
 * Called when a process tried to open the device file, like
 * cat /dev/key_state
 */
static int device_open(struct inode *inode, struct file *file)
{
```

```c
    if (atomic_cmpxchg(&already_open, CDEV_NOT_USED, CDEV_EXCLUSIVE_OPEN))
        return -EBUSY;

    sprintf(msg, static_key_enabled(&fkey) ? "enabled\n" : "disabled\n");

    pr_info("fastpath 1\n");
    if (static_branch_unlikely(&fkey))
        pr_alert("do unlikely thing\n");
    pr_info("fastpath 2\n");

    try_module_get(THIS_MODULE);

    return SUCCESS;
}

/**
 * Called when a process closes the device file
 */
static int device_release(struct inode *inode, struct file *file)
{
    /* We are now ready for our next caller. */
    atomic_set(&already_open, CDEV_NOT_USED);

    /**
     * Decrement the usage count, or else once you opened the file, you will
     * never get rid of the module.
     */
    module_put(THIS_MODULE);

    return SUCCESS;
}

/**
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp, /* see include/linux/fs.h */
                           char __user *buffer, /* buffer to fill with data */
                           size_t length, /* length of the buffer */
                           loff_t *offset)
{
    /* Number of the bytes actually written to the buffer */
    int bytes_read = 0;
    const char *msg_ptr = msg;

    if (!*(msg_ptr + *offset)) { /* We are at the end of the message */
        *offset = 0; /* reset the offset */
        return 0; /* signify end of file */
    }

    msg_ptr += *offset;

    /* Actually put the data into the buffer */
    while (length && *msg_ptr) {
        /**
         * The buffer is in the user data segment, not the kernel
         * segment so "*" assignment won't work. We have to use
```

```
         * put_user which copies data from the kernel data segment to
         * the user data segment.
         */
        put_user(*(msg_ptr++), buffer++);
        length--;
        bytes_read++;
    }

    *offset += bytes_read;

    /* Most read functions return the number of bytes put into the buffer. */
    return bytes_read;
}

/* Called when a process writes to dev file; echo "enable" > /dev/key_state */
static ssize_t device_write(struct file *filp, const char __user *buffer,
                            size_t length, loff_t *offset)
{
    char command[10];

    if (length > 10) {
        pr_err("command exceeded 10 char\n");
        return -EINVAL;
    }

    if (copy_from_user(command, buffer, length))
        return -EFAULT;

    if (strncmp(command, "enable", strlen("enable")) == 0)
        static_branch_enable(&fkey);
    else if (strncmp(command, "disable", strlen("disable")) == 0)
        static_branch_disable(&fkey);
    else {
        pr_err("Invalid command: %s\n", command);
        return -EINVAL;
    }

    /* Again, return the number of input characters used. */
    return length;
}

module_init(chardev_init);
module_exit(chardev_exit);

MODULE_LICENSE("GPL");
```

To check the state of the static key, we can use the /dev/key_state interface.

```
cat /dev/key_state
```

This will display the current state of the key, which is disabled by default.

To change the state of the static key, we can perform a write operation on the file:

```
echo enable > /dev/key_state
```

This will enable the static key, causing the code path to switch from the fastpath to the slowpath.

In some cases, the key is enabled or disabled at initialization and never changed, we can declare a static key as read-only, which means that it can only be toggled in the module init function. To declare a read-only static key, we can use the `DEFINE_STATIC_KEY_FALSE_RO` or `DEFINE_STATIC_KEY_TRUE_RO` macro instead. Attempts to change the key at runtime will result in a page fault. For more information, see Static keys

19 Common Pitfalls

19.1 Using standard libraries

You can not do that. In a kernel module, you can only use kernel functions which are the functions you can see in `/proc/kallsyms`.

19.2 Disabling interrupts

You might need to do this for a short time and that is OK, but if you do not enable them afterwards, your system will be stuck and you will have to power it off.

20 Where To Go From Here?

For those deeply interested in kernel programming, kernelnewbies.org and the Documentation subdirectory within the kernel source code are highly recommended. Although the latter may not always be straightforward, it serves as a valuable initial step for further exploration. Echoing Linus Torvalds' perspective, the most effective method to understand the kernel is through personal examination of the source code.

Contributions to this guide are welcome, especially if there are any significant inaccuracies identified. To contribute or report an issue, please initiate an issue at `https://github.com/sysprog21/lkmpg`. Pull requests are greatly appreciated.

Happy hacking!