

Learning Linux Device Drivers

Jeremiah Mahler (jmmahler@gmail.com)

March 16, 2014

Contents

1	Introduction	2
2	Hello, World	3
2.1	hello	3
2.2	param	4
3	Character Devices	5
3.1	data_chr	5
3.2	data_rw	8
3.3	data_sk	11
3.4	data_ioctl	13
3.5	null, zero	16
3.6	fifo_rw	17
4	Sysfs	20
4.1	fifo_sysfs	20
5	Concurrency	22
5.1	fifo_xxx	22
5.2	fifo_fix	25

1 Introduction

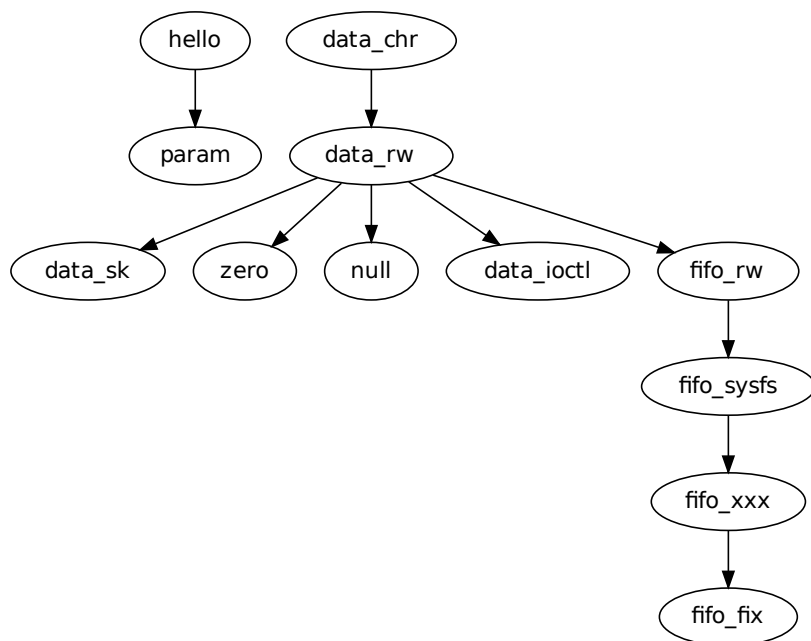


Figure 1: Hierarchy of kernel module examples. Simplest at the top downward to the more complex.

This project is a collection of Linux device driver examples. Most are unique to this project. But many are derived from examples given in books and elsewhere.

A device driver can appear to be complex when taken as a whole. But they are built upon simple concepts. To emphasize this simplicity each driver is built in stages with each stage introducing a new concept. And the changes between stages can be easily viewed by looking at the `'diff'`.

Figure 1 shows the hierarchy of driver examples.

These examples were built using Kernel version 3.9 through 3.13. They will likely work with other versions as well. The Linux kernel changes fast but it usually isn't difficult to determine what has changed and how it can be upgraded.

Each of the modules includes a Makefile to automate the build steps.

```
$ cd hello/  
$ make  
(should compile without error and produce hello.ko)
```

The kernel source and headers will need to be installed if they haven't been already. To accomplish this on one Debian¹ system the following steps were used².

```
$ uname -r  
3.13-1-amd64  
$ sudo apt-get install linux-source linux-headers-amd64
```

¹Debian. *Debian – The Universal Operating System*. [Online; accessed 15-August-2013]. 2013. URL: <http://www.debian.org>.

²It may be necessary to vary these commands for different kernel versions and different Debian releases.

2 Hello, World

2.1 hello

The hello module (Listing 1) simply prints a message when it is loaded/unloaded.

```
hello$ make
(should compile without error, resulting in hello.ko)
hello$ sudo insmod hello.ko
Hello, World
hello$ sudo rmmod hello
Goodbye, cruel world
```

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3
4  static int __init hello_init(void)
5  {
6      printk(KERN_ALERT "Hello , World\n");
7      return 0;
8  }
9
10 static void __exit hello_exit(void)
11 {
12     printk(KERN_ALERT "Goodbye, cruel world\n");
13 }
14
15 MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
16 MODULE_LICENSE("GPL");
17
18 module_init(hello_init);
19 module_exit(hello_exit);
```

Listing 1: Hello, World module in hello/hello.c

The `module_init` (line 18) and `module_exit` (line 19) tell the kernel which functions to call when this module is loaded (`insmod`) and unloaded (`rmmod`).

The `__init` (line 4) and `__exit` (line 10) are optional hints for the compiler. For example, in the case of `__init`, this tells the kernel that it may discard the code after initialization has been completed.

Both the `init` function (line 4) and the `exit` function (line 10) are declared `static`. Since these functions are not meant to be used outside the scope of this file, declaring them `static` enforces this constraint.³

The `printk` statements are the equivalent of a `printf` but in the kernel domain. One difference is that there are various alert levels. In this case `KERN_ALERT` is used which will cause the messages to appear on the console. Notice that there is no comma between the level and the message.

The `MODULE_AUTHOR` and `MODULE_LICENSE` on lines 15 and 16 are optional but recommended.⁴ There are various other `MODULE_*` options as well (`linux/module.h`).

³J. Corbet, A. Rubini, and Greg. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 2009. ISBN: 9780596555382, Pg. 52.

⁴Ibid., Pg. 51.

2.2 param

The param module expands upon the hello module to take a parameter specifying how many times to print the message.

```
param$ sudo insmod hello.ko howmany=2
Hello, World
Hello, World
param$ sudo rmmod hello
Goodbye, cruel world
Goodbye, cruel world
```

Listing 2 shows the differences between this parameterized hello world module and the previous hello module.

```
1  --- ../hello/hello.c      2013-08-09 12:23:58.222416131 -0700
2  +++ hello.c 2013-08-09 12:53:38.082434726 -0700
3  @@ -1,15 +1,28 @@
4  #include <linux/init.h>
5  #include <linux/module.h>
6  +#include <linux/moduleparam.h>
7  +
8  +static int howmany = 1;
9  +module_param(howmany, int, S_IRUGO);
10
11  static int __init hello_init(void)
12  {
13  -   printk(KERN_ALERT "Hello , World\n");
14  +   int i;
15  +
16  +   for (i = 0; i < howmany; i++) {
17  +       printk(KERN_ALERT "Hello , World\n");
18  +   }
19  +
20  +   return 0;
21  }
22
23  static void __exit hello_exit(void)
24  {
25  -   printk(KERN_ALERT "Goodbye, cruel world\n");
26  +   int i;
27  +
28  +   for (i = 0; i < howmany; i++) {
29  +       printk(KERN_ALERT "Goodbye, cruel world\n");
30  +   }
31  }
32
33  MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
```

Listing 2: param\$ diff -u hello.c ../hello/hello.c

To use a parameter a global variable has been created named `howmany` on line 8. And on line 9 the `module_param` function is used to tell the kernel about this parameter ⁵.

On lines 13-19 and 25-30 it can be seen that the same message is printed `howmany` times.

⁵The `module_param` function create a `sysfs` entry in `/sys/module/parameters/howmany`. `sysfs` will be discussed in detail in later modules.

3 Character Devices

The `data_*` modules allocate a memory area which can be read from and written to. This is accomplished as a character device which will support the usual file operations (read, write, seek).

3.1 `data_chr`

The first step is to construct the basic infrastructure for a character driver as shown in Listing 3. This code is significantly larger than the `hello` driver (Section 2.1) and it doesn't even provide any of the read, write or seek operations. But it serves as a base from which subsequent drivers will be built from.

The `DEVICE_NAME` (line 8) is just a shortcut for the name which is used in several places.

Lines 10-17 are the global variables that will be used. The `struct data_dev` is the per device structure. Notice that a character device is placed inside.

The `file_operations` (line 18-20) defines just the `.owner` value. Later modules will add references to the open, close, read, write, and seek functions here.

The `alloc_chrdev_region` call (line 26) allocates a major and minor number for the character device.⁶ In this case only one major and minor pair is needed.

Functions such as `alloc_chrdev_region` may fail. And when they do anything that has been created up to that point must be undone to ensure the kernel is kept in a consistent state. A common way this is done is using `goto` statements which branches to different steps in the exit sequence.⁷ It can be seen that if `alloc_chrdev_region` fails, its `goto` (line 29) will branch to line 66. Since nothing was created up to that point nothing has to be undone.

`class_create` (line 32) establishes a “class” for this module which create a representation in `sysfs` under `/sys/class/data`. This object will be used later as an argument to `device_create`.

Since the per device structure is just a pointer (line 16) it must be allocated before it is used (line 34).

To establish the character device it must be initialized and added (lines 41-43). And finally the device is created (line 49). This device will now appear under `/dev/data0`.

⁶Corbet, Rubini, and Kroah-Hartman, see n. 3, Pg. 66.

⁷Ibid., Pg. 53.

```

1  #include <linux/cdev.h>
2  #include <linux/device.h>
3  #include <linux/fs.h>
4  #include <linux/module.h>
5  #include <linux/slab.h>
6  #include <linux/uaccess.h>
7
8  #define DEVICENAME "data"
9
10 static dev_t data_major;
11 struct class *data_class;
12 struct device *data_device;
13
14 struct data_dev {
15     struct cdev cdev;
16 } *data_devp;
17
18 struct file_operations data_fops = {
19     .owner = THIS_MODULE,
20 };
21
22 static int __init data_init(void)
23 {
24     int err = 0;
25
26     err = alloc_chrdev_region(&data_major, 0, 1, DEVICENAME);
27     if (err < 0) {
28         printk(KERN_WARNING "Unable to register device\n");
29         goto err_chrdev_region;
30     }
31
32     data_class = class_create(THIS_MODULE, DEVICENAME);
33
34     data_devp = kmalloc(sizeof(struct data_dev), GFP_KERNEL);
35     if (!data_devp) {
36         printk(KERN_WARNING "Unable to kmalloc data_devp\n");
37         err = -ENOMEM;
38         goto err_malloc_data_devp;
39     }
40
41     cdev_init(&data_devp->cdev, &data_fops);
42     data_devp->cdev.owner = THIS_MODULE;
43     err = cdev_add(&data_devp->cdev, data_major, 1);
44     if (err) {
45         printk(KERN_WARNING "cdev_add failed\n");
46         goto err_cdev_add;
47     }
48
49     data_device = device_create(data_class, NULL,
50                               MKDEV(MAJOR(data_major), 0), NULL, "data%d", 0);
51     if (IS_ERR(data_device)) {
52         printk(KERN_WARNING "device_create failed\n");
53         err = PTR_ERR(data_device);
54         goto err_device_create;
55     }
56
57     return 0;  /* success */
58
59 err_device_create:
60     cdev_del(&data_devp->cdev);
61 err_cdev_add:
62     kfree(data_devp);
63 err_malloc_data_devp:
64     class_destroy(data_class);
65     unregister_chrdev_region(data_major, 1);
66 err_chrdev_region:
67
68     return err;
69 }

```

```

70
71 static void __exit data_exit(void)
72 {
73     device_destroy(data_class, data_major);
74     cdev_del(&data_devp->cdev);
75     kfree(data_devp);
76     class_destroy(data_class);
77     unregister_chrdev_region(data_major, 1);
78 }
79
80 MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
81 MODULE_LICENSE("GPL");
82
83 module_init(data_init);
84 module_exit(data_exit);
85

```

Listing 3: data_chr driver.

3.2 data_rw

The `data_rw` driver adds read/write operations to the core `data_chr` driver (Section 3.1).

With the addition of read/write operations the device can be operated upon just like any other file. As an example, the driver source code can be copied in to the device and then read back out. The result should be the same, up to a maximum amount, which in this case is 128 bytes⁸.

```
$ sudo dd if=data.c of=/dev/data0 bs=128 count=1
```

```
$ sudo dd if=/dev/data0 of=output bs=128 count=1
```

Listing 4 shows the differences between this driver and `data_chr` driver.

An array of bytes has been added to store the data (line 15). Notice that it is inside the per device structure. A file offset (line 16) has also been added.

File operations for open, read, write and release have been added (lines 19, 31, 55). The release operation (line 79) is called when a process closes the device file.

When the file is opened the open function (lines 19-29) is called. The `container_of` function (line 23) is used to obtain a parent structure from a child structure^{9,10}. Recall that the per device structure, `data_devp`, contains a `cdev` (line 14). `container_of` makes it possible to obtain the `data_devp` from the `cdev`.

The open function sets the file offset to zero (line 24) when the file is opened. This is the usual behavior when opening a file.

The open function also stores the device structure under `private_data` (line 26) so it is easy to access in the read/write functions.

When data is read from the device file the read function (line 31) is called. Since the amount of data that can be read is limited by `MAX_DATA` the amount requested will be reduced if it is too large (lines 40-42). Then the `copy_to_user` function is used to attempt to transfer the data to user space (line 46). The `copy_to_user` function will also check to make sure the destination it is transferring to is valid for the given process. If the transfer was a success the new offset is stored (line 50) and then the number of bytes that were successfully transferred are returned (line 52).

The write function (line 55) is largely the same as the read function but in the opposite direction. Notice that the `copy_from_user` (line 70) function is used instead of `copy_to_user`.

Since nothing needs to be done when the device is closed, the release function (line 79) simply returns success.

⁸The maximum data size is defined inside the driver.

⁹Corbet, Rubini, and Kroah-Hartman, see n. 3, Pg. 79.

¹⁰Greg. Kroah-Hartman. *container_of()*. [Online; accessed 10-August-2013]. 2005. URL: http://www.kroah.com/log/linux/container_of.html.

```

1  --- ../data_chr/data.c 2013-08-10 10:17:01.359016284 -0700
2  +++ data.c 2013-08-16 21:55:03.833816746 -0700
3  @@ -6,6 +6,7 @@
4  #include <linux/uaccess.h>
5
6  #define DEVICE_NAME "data"
7  +#define MAX_DATA 128
8
9  static dev_t data_major;
10 struct class *data_class;
11 @@ -13,10 +14,81 @@
12
13 struct data_dev {
14     struct cdev cdev;
15 +   char data[MAX_DATA];
16 +   loff_t cur_ofs; // current offset
17 } *data_devp;
18
19 +static int data_open(struct inode* inode, struct file* filp)
20 +{
21 +   struct data_dev *data_devp;
22 +
23 +   data_devp = container_of(inode->i_cdev, struct data_dev, cdev);
24 +   data_devp->cur_ofs = 0;
25 +
26 +   filp->private_data = data_devp;
27 +
28 +   return 0;
29 +}
30 +
31 +static ssize_t data_read(struct file *filp, char __user *buf,
32 +                        size_t count,
33 +                        loff_t *f_pos)
34 +{
35 +   struct data_dev *data_devp = filp->private_data;
36 +   loff_t cur_ofs;
37 +   char *datap;
38 +   size_t left;
39 +
40 +   cur_ofs = data_devp->cur_ofs;
41 +   datap = data_devp->data;
42 +   left = MAX_DATA - cur_ofs;
43 +
44 +   count = (count > left) ? left : count;
45 +
46 +   if (copy_to_user(buf, (void *) (datap + cur_ofs), count) != 0) {
47 +       return -EIO;
48 +   }
49 +
50 +   data_devp->cur_ofs = cur_ofs + count;
51 +
52 +   return count;
53 +}
54 +
55 +static ssize_t data_write(struct file *filp, const char __user *buf,
56 +                          size_t count,
57 +                          loff_t *f_pos)
58 +{
59 +   struct data_dev *data_devp = filp->private_data;
60 +   loff_t cur_ofs;
61 +   char *datap;
62 +   size_t left;
63 +
64 +   cur_ofs = data_devp->cur_ofs;
65 +   datap = data_devp->data;
66 +   left = MAX_DATA - cur_ofs;
67 +
68 +   count = (count > left) ? left : count;
69 +

```

```

70 +   if (copy_from_user((void *) (datap + cur_ofs), buf, count) != 0) {
71 +       return -EIO;
72 +   }
73 +
74 +   data_devp->cur_ofs = cur_ofs + count;
75 +
76 +   return count;
77 +}
78 +
79 +static int data_release(struct inode *inode, struct file *filp)
80 +{
81 +   return 0;
82 +}
83 +
84 +struct file_operations data_fops = {
85 +   .owner = THIS_MODULE,
86 +   .open = data_open,
87 +   .read = data_read,
88 +   .write = data_write,
89 +   .release = data_release,
90 +};
91
92 +static int __init data_init(void)
93 @@ -35,7 +107,7 @@
94 +   if (!data_devp) {
95 +       printk(KERN_WARNING "Unable to kmalloc data_devp\n");
96 +       err = -ENOMEM;
97 -       goto err_malloc_data_devp;
98 +       goto err_malloc_devp;
99 +   }
100
101 +   cdev_init(&data_devp->cdev, &data_fops);
102 @@ -60,7 +132,7 @@
103 +   cdev_del(&data_devp->cdev);
104 +   err_cdev_add:
105 +   kfree(data_devp);
106 -err_malloc_data_devp:
107 +err_malloc_devp:
108 +   class_destroy(data_class);
109 +   unregister_chrdev_region(data_major, 1);
110 +   err_chrdev_region:

```

Listing 4: data_rw\$ diff -u ../data_chr/data.c data.c

3.3 data_sk

To add support for the seek operation one more function must be added along with its corresponding entry in the file operations structure. Listing 5 shows the differences.

Checking the seek operation requires only a few steps. First, write some random data to the device.

```
~/l1dd/data_sk/test$ sudo dd if=/dev/urandom of=/dev/data0 bs=128 count=1
1+0 records in
1+0 records out
128 bytes (128 B) copied, 0.000542457 s, 236 kB/s
```

To check the seek operation, read the data in two different ways. From the beginning and then seeking forward 100 bytes. From the end and then seeking back -28 bytes. The `cats` program is a seek-able version of `cat` which is included with the driver.

```
~/l1dd/data_sk/test$ sudo ./cats /dev/data0 SET 100 > out1.dat
~/l1dd/data_sk/test$ sudo ./cats /dev/data0 END -28 > out2.dat
```

If seek operation is working properly the data read should be the same in both cases.

```
~/l1dd/data_sk/test$ md5sum out1.dat out2.dat
e50c04e81ad918d121960932f98c95b3 out1.dat
e50c04e81ad918d121960932f98c95b3 out2.dat
```

```
1  --- ../data_rw/data.c  2013-08-16 21:55:03.833816746 -0700
2  +++ data.c  2013-08-16 21:56:15.829814250 -0700
3  @@ -78,6 +78,35 @@
4      return count;
5  }
6
7  +static loff_t data_llseek(struct file *filp, loff_t offset, int orig)
8  +{
9  +    struct data_dev *data_devp = filp->private_data;
10 +    loff_t cur_ofs;
11 +
12 +    cur_ofs = data_devp->cur_ofs;
13 +
14 +    switch (orig) {
15 +        case SEEK_SET:
16 +            cur_ofs = offset;
17 +            break;
18 +        case SEEK_CUR:
19 +            cur_ofs += offset;
20 +            break;
21 +        case SEEK_END:
22 +            cur_ofs = MAXDATA + offset;
23 +            break;
24 +        default:
25 +            return -EINVAL;
26 +    }
27 +
28 +    if (cur_ofs < 0 || cur_ofs >= MAXDATA)
29 +        return -EINVAL;
30 +
31 +    data_devp->cur_ofs = cur_ofs;
32 +
33 +    return cur_ofs;
34 +}
35 +
36  static int data_release(struct inode *inode, struct file *filp)
37  {
38      return 0;
39  @@ -88,6 +117,7 @@
40      .open = data_open,
41      .read = data_read,
42      .write = data_write,
43  +    .llseek = data_llseek,
```

```

44     .release = data_release ,
45 };
46
47 @@ -107,7 +137,7 @@
48     if (!data_devp) {
49         printk(KERN_WARNING "Unable to kmalloc data_devp\n");
50         err = -ENOMEM;
51 -        goto err_malloc_devp;
52 +        goto err_malloc_data_devp;
53     }
54
55     cdev_init(&data_devp->cdev, &data_fops);
56 @@ -132,7 +162,7 @@
57     cdev_del(&data_devp->cdev);
58     err_cdev_add:
59     kfree(data_devp);
60 -err_malloc_devp:
61 +err_malloc_data_devp:
62     class_destroy(data_class);
63     unregister_chrdev_region(data_major, 1);
64     err_chrdev_region:

```

Listing 5: data_sk\$ diff -u ../data_rw/data.c data.c

3.4 data_ioctl

The `data_ioctl` driver shows how support for `ioctl()` can be added to a driver. Note that using `ioctl()` for new designs is not recommended^{11,12}. It is recommended to use `sysfs` instead. While `/proc` is another option, it is also becoming obsolete in favor of `sysfs`. Nonetheless, it is still worth knowing because older drivers may still use this operation.

The driver allows the reset, read and write of a single global variable (`x`) in the driver. Included with this driver is a test program (`data_ioctl/test/ioctlx.c`) which can be used for testing these operations.

```
data_ioctl$ cd test/
test$ sudo ./ioctlx 10      # set value
test$ sudo ./ioctlx        # read current value
10
test$ sudo ./ioctlx 0      # reset
test$ sudo ./ioctlx
0
test$
```

The `data_ioctl` driver has the fewest number of differences compared to the `data_chr` driver (Section 3.1) as shown in Listing 6. The open (lines 10-19) and release (lines 75-78) are the same as those for the `data_rw` driver (Section 3.2). The only code of interest is for the `data_ioctl` function (lines 29-73).

The first thing to notice is the use of "magic" (lines 23-25, 37). Magic is used to make the `ioctl` calls unique across the entire system which helps prevent inadvertent configuration if the wrong device is opened.¹³ The user program must also contain the corresponding magic values as is done in the test program (Listing 7).

The three defines (lines 23-25) describe the supported `ioctl` operations. Any number of additional operations can be added. There are operations for reading data (`_IOR`), writing data (`_IOW`), and commands with no data (`_IO`). The `DATA_IOC_MAXNR` (line 27) is used as a sanity check later in the code (line 41).

The switch statement (lines 53-70) processes the `ioctl` commands. In this case all the operations involve the global variable `x`. It is either reset, read or written.

¹¹Corbet, Rubini, and Kroah-Hartman, see n. 3, Pg. 156.

¹²R. Love. *Linux Kernel Development*. Developer's Library. Pearson Education, 2010. ISBN: 9780768696790.

¹³Corbet, Rubini, and Kroah-Hartman, see n. 3, Pg. 158.

```

1  --- ../data_chr/data.c 2013-08-10 10:17:01.359016284 -0700
2  +++ data.c 2013-08-11 11:08:06.287053188 -0700
3  @@ -15,14 +15,92 @@
4      struct cdev cdev;
5      } *data_devp;
6
7  +
8  +int x;
9  +
10 +static int data_open(struct inode *inode, struct file *filp)
11 +{
12 +    struct data_dev *data_devp;
13 +
14 +    data_devp = container_of(inode->i_cdev, struct data_dev, cdev);
15 +
16 +    filp->private_data = data_devp;
17 +
18 +    return 0;
19 +}
20 +
21 +#define DATA_IOC_MAGIC 'm'
22 +
23 +#define DATA_IOCRESET _IO(DATA_IOC_MAGIC, 1)
24 +#define DATA_IOCWX _IOW(DATA_IOC_MAGIC, 2, int)
25 +#define DATA_IOCRX _IOR(DATA_IOC_MAGIC, 3, int)
26 +
27 +#define DATA_IOC_MAXNR 3
28 +
29 +static long data_ioctl(struct file *filp, unsigned int cmd,
30 +                      unsigned long arg)
31 +{
32 +    int err = 0;
33 +    int retval = 0;
34 +
35 +    //struct data_dev *data_dev = filp->private_data;
36 +
37 +    if (_IOC_TYPE(cmd) != DATA_IOC_MAGIC) {
38 +        printk(KERN_ALERT "invalid ioctl magic\n");
39 +        return -ENOTTY;
40 +    }
41 +    if (_IOC_NR(cmd) > DATA_IOC_MAXNR) {
42 +        printk(KERN_ALERT "ioctl beyond maximum\n");
43 +        return -ENOTTY;
44 +    }
45 +
46 +    if (_IOC_DIR(cmd) & _IOC_READ)
47 +        err = !access_ok(VERIFY_WRITE, (void __user *) arg, _IOC_SIZE(cmd));
48 +    else if (_IOC_DIR(cmd) & _IOC_WRITE)
49 +        err = !access_ok(VERIFY_READ, (void __user *) arg, _IOC_SIZE(cmd));
50 +    if (err)
51 +        return -EFAULT;
52 +
53 +    switch (cmd) {
54 +        case DATA_IOCRESET:
55 +            /* takes no argument, sets values to default */
56 +            x = 0;
57 +            break;
58 +        case DATA_IOCRX:
59 +            /* read integer */
60 +            retval = __put_user(x, (int __user *) arg);
61 +            break;
62 +        case DATA_IOCWX:
63 +            /* write integer */
64 +            retval = __get_user(x, (int __user *) arg);
65 +            break;
66 +        default:
67 +            return -ENOTTY; /* POSIX standard */
68 +            //return -EINVAL; /* common */
69 +            /* Pg. 161 Linux Device Drivers (2005) */

```

```

70 +   }
71 +
72 +   return retval;
73 +}
74 +
75 +int data_release(struct inode *inode, struct file *filp)
76 +{
77 +   return 0;
78 +}
79 +
80 +   struct file_operations data_fops = {
81 +       .owner = THIS_MODULE,
82 +       .open = data_open,
83 +       .unlocked_ioctl = data_ioctl,
84 +       .release = data_release,
85 +   };
86 +
87 +   static int __init data_init(void)
88 +   {
89 +       int err = 0;
90 +
91 +       x = 0;
92 +
93 +       err = alloc_chrdev_region(&data_major, 0, 1, DEVICENAME);
94 +       if (err < 0) {
95 +           printk(KERN_WARNING "Unable to register device\n");

```

Listing 6: data_ioctl\$ diff -u ../data_chr/data.c data.c

```

9  #define DEVFILE "/dev/data0"
10
11 #define DATA_IOC_MAGIC 'm'
12
13 #define DATA_IOCRESET _IO(DATA_IOC_MAGIC, 1)
14 #define DATA_IOCWX    _IOW(DATA_IOC_MAGIC, 2, int)
15 #define DATA_IOCRX    _IOR(DATA_IOC_MAGIC, 3, int)
16
17 int main(int argc, char* argv[])
18 {
19     int devfd;

```

Listing 7: Corresponding "magic" in user program.

3.5 null, zero

Most Linux users are probably familiar with `/dev/null` and `/dev/zero`. The first discards and data written to it and the second returns zero for any read. A driver for these devices can be easily constructed from what has been discussed so far about character devices. In fact, they are probably simpler than the `data_rw` driver (Section 3.2).

The only real difference, other than names (`data -> null`), is the read and write operations as shown in Listing 8.

```
30 static ssize_t null_read(struct file *filp, char __user *buf,
31                          size_t count, loff_t *f_pos)
32 {
33     return 0;
34 }
35
36 static ssize_t null_write(struct file *filp, const char __user *buf,
37                          size_t count, loff_t *f_pos)
38 {
39     return count;
40 }
```

Listing 8: `/dev/null` read and write functions.

The read and write functions for the `zero` driver are also quite simple (Listing 9). The one new addition is the `clear_user` function. It behaves like the `copy_to_user` function except that it simply zeros out the users buffer.

```
30 static ssize_t zero_read(struct file *filp, char __user *buf,
31                          size_t count, loff_t *f_pos)
32 {
33     if (clear_user((void __user *) buf, count) > 0) {
34         return -EFAULT;
35     }
36
37     return count;
38 }
39
40 ssize_t zero_write(struct file *filp, const char __user *buf,
41                   size_t count, loff_t *f_pos)
42 {
43     return count;
44 }
```

Listing 9: `/dev/zero` read and write functions.

3.6 fifo_rw

A FIFO (first in first out) can be constructed as a character device. And it requires no new techniques beyond what was described for the `data_rw` driver (Section 3.2).

Several test programs are included (`test/`) to simplify experimenting with the fifo device. The `fifor` and `fifow` programs can be used to read and write numbers to the device.

The following example writes four numbers to the fifo and then reads them back out.

```
test$ sudo ./fifow 10 11 12 13
test$ sudo ./fifor 4
10
11
12
test$
```

Notice that only three values could be read out. This is because by default fifo size is three (`#define MAX_DATA`). Once the fifo is full it discards any other values.

The code for the `fifo_rw` is largely the same as the `data_rw` driver (Section 3.2). The fifo is built using read and write pointers along with an empty flag. When the module is initialized these values are set (Listing 10).

```
138     fifo_devp->fifo_start = &fifo_devp->fifo[0];
139     fifo_devp->fifo_end   = &fifo_devp->fifo[MAX_DATA-1];
140     fifo_devp->read_ptr   = &fifo_devp->fifo[0];
141     fifo_devp->write_ptr  = &fifo_devp->fifo[0];
142     fifo_devp->empty      = 1;
```

Listing 10: `fifo_rw` driver init.

The read and write functions (Listing 11 and 12) have only algorithmic differences compared to the same operations in the `data_rw` driver.

The fifo works a byte at a time trying to read/write data until there is no more data or there is no more room left in the fifo. With this fifo design it is either empty or full if the read pointer is at the same position as the write pointer. To distinguish between these cases an empty flag is used. Any successful write will make empty false (Listing 12, lines 90-91). Any successful read, which results in the read pointer being the same as the write pointer, will make empty true (Listing 11, lines 62-64).

```

36 static ssize_t fifo_read(struct file *filp, char __user *buf,
37                          size_t count,
38                          loff_t *f_pos)
39 {
40     struct fifo_dev *dev = filp->private_data;
41     size_t left;
42
43     left = count;
44
45     while (left) {
46
47         if (dev->empty) {
48             break;
49         }
50
51         if (copy_to_user(buf, (void *) dev->read_ptr, 1) != 0) {
52             return -EIO;
53         }
54         left--;
55
56         if (dev->read_ptr == dev->fifo_end) {
57             dev->read_ptr = dev->fifo_start;
58         } else {
59             (dev->read_ptr)++;
60         }
61
62         if (dev->read_ptr == dev->write_ptr) {
63             dev->empty = 1;
64         }
65     }
66
67     return (count - left);
68 }

```

Listing 11: fifo_rw driver read function.

```

70 static ssize_t fifo_write(struct file *filp, const char __user *buf,
71                          size_t count,
72                          loff_t *f_pos)
73 {
74     struct fifo_dev *dev = filp->private_data;
75     size_t left;
76
77     left = count;
78
79     while (left) {
80
81         if (!(dev->empty) && (dev->read_ptr == dev->write_ptr)) {
82             break;
83         }
84
85         if (copy_from_user((void *) dev->write_ptr, buf, 1) != 0) {
86             return -EIO;
87         }
88         left--;
89
90         if (dev->empty)
91             dev->empty = 0;
92
93         if (dev->write_ptr == dev->fifo_end) {
94             dev->write_ptr = dev->fifo_start;
95         } else {
96             (dev->write_ptr)++;
97         }
98     }
99
100     return (count - left);
101 }

```

Listing 12: fifo_rw driver write function.

4 Sysfs

4.1 fifo_sysfs

With the `fifo_rw` (Section 3.6) driver it was possible to write values and read them back out in a first in first out order. But there was no way to determine how many values were in the fifo or how many spaces were left.

One way to create access to these values is by using Sysfs attributes. Sysfs is a file system representation of all the objects in the kernel. An “attribute” can be added to a kernel object (kobject) and that attribute can appear as a file in `/sys/`. And this allows the value to read from or written to.

To add an attribute the first thing that must be done is to add the operations for read and write. A common sysfs convention is to name these operations *show* and *store* instead of read and write (Listing 13).

The `DEVICE_ATTR` macro is used to create a `dev_attr` structure which will be used later when creating the file. In this example `read_offset` (line 134) will create a variable named `dev_attr_read_offset`.

```
117 static ssize_t read_offset_show(struct device *dev,
118                                struct device_attribute *attr,
119                                char *buf)
120 {
121     struct fifo_dev *fifo_devp = dev_get_drvdata(dev);
122     return sprintf(buf, "%u\n", (unsigned int) (fifo_devp->read_ptr
123                                                - fifo_devp->fifo_start));
124 }
125
126 static ssize_t read_offset_store(struct device *dev,
127                                 struct device_attribute *attr,
128                                 const char *buf,
129                                 size_t count)
130 {
131     return 0; // cannot store anything
132 }
133
134 static DEVICE_ATTR(read_offset, 0666, read_offset_show, read_offset_store);
135
136 static ssize_t write_offset_show(struct device *dev,
137                                  struct device_attribute *attr,
138                                  char *buf)
139 {
140     struct fifo_dev *fifo_devp = dev_get_drvdata(dev);
141     return sprintf(buf, "%u\n", (unsigned int) (fifo_devp->write_ptr
142                                                - fifo_devp->fifo_start));
143 }
144
145 static ssize_t write_offset_store(struct device *dev,
146                                   struct device_attribute *attr,
147                                   const char *buf,
148                                   size_t count)
149 {
150     return 0; // cannot store anything
151 }
152
153 static DEVICE_ATTR(write_offset, 0666, write_offset_show, write_offset_store);
```

Listing 13: `fifo_sysfs` driver show/store.

There are many different kernel structures and attributes. They each have their own attribute macro similar to `DEVICE_ATTR` such as: `DRIVER_ATTR`, `CLASS_ATTR`, `BUS_ATTR`, etc. Refer to `linux/device.h` for more info.

Adding an attribute to a kernel object does not necessarily create a corresponding file in `/sys/`. To do this the `device_create_file()` function must be used (Listing 14). Notice that the `dev_attr_*` variable that was created earlier is used along with `fifo_device` structure.

```
203     err = device_create_file(fifo_device , &dev_attr_read_offset);
204     if (err) {
205         goto err_file_read_offset;
206     }
207
208     err = device_create_file(fifo_device , &dev_attr_write_offset);
209     if (err) {
210         goto err_file_write_offset;
211     }
```

Listing 14: `fifo_sysfs` driver attribute file creation.

One thing that takes getting used to with Sysfs and `kobject`'s is where things end up under `/sys/`. It is often best to add the attributes to the most relevant object. Then they will usually end up in a place that makes sense. Trying to find the object that will place the file where you want is often the wrong approach.

In this example the attributes end up under:

```
fifo_sysfs$ sudo find /sys -name 'read_offset'
/sys/devices/virtual/fifo/fifo0/read_offset
fifo_sysfs$ ls /sys/devices/virtual/fifo/fifo0
dev power read_offset subsystem uevent write_offset
fifo_sysfs$
```

Notice that both attributes were created there. Several extra attributes were also created automatically.

Using these new attributes the test programs from the `fifo_rw` driver can be used to verify its operation.

```
test$ cat /sys/devices/virtual/fifo/fifo0/read_offset
0
test$ cat /sys/devices/virtual/fifo/fifo0/write_offset
0
test$ sudo ./fifow 10 11
test$ cat /sys/devices/virtual/fifo/fifo0/read_offset
0
test$ cat /sys/devices/virtual/fifo/fifo0/write_offset
2
test$ sudo ./fifor 1
10
test$ cat /sys/devices/virtual/fifo/fifo0/read_offset
1
test$ cat /sys/devices/virtual/fifo/fifo0/write_offset
2
```

As expected, a write will increment the write offset by the number of values written. And a read will increase the read pointer by the number of values read.

5 Concurrency

For the sake of simplicity the examples up to this point have ignored problems related to concurrency.

When multiple processes access a shared resource without restriction (such as with mutexes or semaphores) the resulting race conditions can cause it to behave unexpectedly. For example, if a pointer offset was incremented beyond its maximum a write could crash the system.

Even if multiple processes access a resource the negative effects of race conditions may still be infrequent and difficult to reproduce. The technique used here is to first add code which guarantees the negative will occur. Then code is added to fix the problems.

5.1 fifo_xxx

To magnify the effects of race conditions in the fifo driver a pwait() function has been created. This function forces the driver to wait for two processes to arrive at a particular point before allowing them both to proceed on. Listing 15 shows the additions.

```
47 +DEFINE_PWAIT(fifo_read);
48 +
49  static ssize_t fifo_read(struct file *filp, char __user *buf,
50                          size_t count,
51                          loff_t *f_pos)
52 @@ -44,6 +78,7 @@
53
54     while (left) {
55
56 +        pwait_fifo_read();
57         if (dev->empty) {
58             break;
59         }
60 @@ -67,6 +102,8 @@
61     return (count - left);
62 }
63
64 +DEFINE_PWAIT(fifo_write);
65 +
66  static ssize_t fifo_write(struct file *filp, const char __user *buf,
67                          size_t count,
68                          loff_t *f_pos)
69 @@ -78,6 +115,8 @@
70
71     while (left) {
72
73 +        pwait_fifo_write();
74 +
75         if (!(dev->empty) && (dev->read_ptr == dev->write_ptr)) {
76             break;
77         }
```

Listing 15: Added pwait_*() to fifo.c driver to create race conditions.

While the addition of pwait_*() certainly magnifies the negative effects of race conditions it still has a degree of unpredictability. Inconsistent results are easily produced but the actual results vary.

In the following example two terminals are used to read and write values. From terminal 1 the values 10 and 11 are written. And at the same time terminal 2 writes the value 20 and 21. Then they both try to read out all three of the stored values.

```
test$ cat /sys/devices/virtual/fifo/fifo0/read_offset
0
test$ cat /sys/devices/virtual/fifo/fifo0/write_offset
0

test$ # (terminal 1)
test$ sudo ./fifow 10

test$ # (terminal 2)
test$ sudo ./fifow 20

test$ # (terminal 1)
test$ sudo ./fifow 11

test$ # (terminal 2)
test$ sudo ./fifow 21

test$ # (terminal 1)
test$ sudo ./fifor 3
10
20

test$ # (terminal 2)
test$ sudo ./fifor 3
20
11

test$ cat /sys/devices/virtual/fifo/fifo0/read_offset
1
test$ cat /sys/devices/virtual/fifo/fifo0/write_offset
0

test$ # (terminal 1)
test$ sudo ./fifor 3
test$ # (terminal 2)
test$ sudo ./fifor 3
test$ # (empty?)
```

It should be apparent that this driver has concurrency problems. Four values were read out in total which is impossible since the fifo size is three. The value 20 was read twice. And this reading of three values should have emptied the fifo yet the offsets are not equal which indicates that it is not empty. Attempting to read more values indicates that it is empty. The fifo should never have the empty flag set with non-equal read and write offsets.

The code that is used to create these race conditions is interesting because it uses mutexes (Listing 16). Mutexes will be used to fix this code so that it doesn't suffer from race conditions.

The first thing to notice about the code (Listing 16) is that it is a macro with a uid substitution. Every place which has a `##uid` will be substituted with the given unique id. This uniqueness is needed to create unique mutexes. This macro will create a function which can then be called. For example `DEFINE_PWAIT(read_fifo)` results in the function `pwait_read_fifo()`.

The code works by using the variables `in` and `out`. Each process that enters the code increments `in`.

When `in` reaches two, two `ins` are removed and two `outs` are added indicating that two process may leave. Any process that leaves decrements `out`.

Mutexes are used to limit access to each of the `in` and `out` variables to only one process. The code on lines 57-65 is an interesting example. At first glance it may appear that the `out` mutex could be unlocked twice. But in fact the inner unlock is necessary because the `break` would skip the outer unlock. Forgetting the inner unlock would result in a deadlock since that lock would never be unlocked.

```

39 #define DEFINE_PWAIT(uid)                                     \
40                                                                    \
41 static DEFINE_MUTEX(in_mtx_##uid);                             \
42 static DEFINE_MUTEX(out_mtx_##uid);                             \
43                                                                    \
44 static void pwait_##uid(void) {                                  \
45     static int in = 0;                                           \
46     static int out = 0;                                           \
47                                                                    \
48     mutex_lock(&in_mtx_##uid);                                    \
49     if (++in >= 2) {                                              \
50         mutex_lock(&out_mtx_##uid);                              \
51         out += 2;                                                 \
52         mutex_unlock(&out_mtx_##uid);                            \
53         in -= 2;                                                  \
54     }                                                            \
55     mutex_unlock(&in_mtx_##uid);                                  \
56                                                                    \
57     do {                                                         \
58         mutex_lock(&out_mtx_##uid);                              \
59         if (out) {                                                \
60             out -= 1;                                             \
61             mutex_unlock(&out_mtx_##uid);                        \
62             break;                                                \
63         }                                                        \
64         mutex_unlock(&out_mtx_##uid);                            \
65     } while (1);                                                 \
66 }

```

Listing 16: `pwait` function used to create race conditions in `fifo_xxx` driver.

5.2 fifo_fix

The race conditions that ruined the operation of the `fifo_xxx` can be fixed using a single mutex. In other situations it might be beneficial to make the control more fine grained with multiple mutexes. But in this example simplicity was important.

The first change that was made was to add a mutex to the device structure (Listing 17). Some may wonder why this mutex was made device specific instead of global. In this simple example either way would work. But it is common to have multiple devices in which case it would be undesirable for a mutex on one device to inhibit the operation of some other device.

```
1  --- ../fifo_xxx/fifo.c 2013-08-17 10:06:23.329527514 -0700
2  +++ fifo.c 2013-08-17 10:06:49.457199045 -0700
3  @@ -23,6 +23,7 @@
4      char *fifo_start;
5      char *fifo_end;
6      int empty;
7  + struct mutex ferw_mtx; // ferw = fifo, empty, read, write
8  } *fifo_devp;
```

Listing 17: Mutex added to device structure.

The code added to the read and write operations are nearly identical. Listing 18 shows the read operation differences. The mutex is locked just before the code which must read or write the shared variables. And the mutex must be unlocked before this scope is exited.

```
11 @@ -79,11 +80,16 @@
12     while (left) {
13
14         pwait_fifo_read();
15     +
16     +     mutex_lock(&dev->ferw_mtx);
17     +
18         if (dev->empty) {
19     +     mutex_unlock(&dev->ferw_mtx);
20         break;
21     }
22
23     if (copy_to_user(buf, (void *) dev->read_ptr, 1) != 0) {
24     +     mutex_unlock(&dev->ferw_mtx);
25     return -EIO;
26     }
27     left--;
28 @@ -97,6 +103,7 @@
29     if (dev->read_ptr == dev->write_ptr) {
30         dev->empty = 1;
31     }
32     +     mutex_unlock(&dev->ferw_mtx);
33     }
34
35     return (count - left);
```

Listing 18: Mutex added to read operation. Write operation is similar

And during initialization the mutex must be initialized (Listing 19). In contrast, when a global mutex is initialized this is often accomplished as part of `DEFINE_MUTEX`.

```

60 @@ -220,6 +232,8 @@
61     fifo_devp->write_ptr = &fifo_devp->fifo[0];
62     fifo_devp->empty = 1;
63
64 +   mutex_init(&fifo_devp->ferw_mtx);
65 +
66     err = cdev_add(&fifo_devp->cdev, fifo_major, 1);
67     if (err) {
68         printk(KERN_WARNING "cdev_add failed\n");

```

Listing 19: Mutex initialization.

An example session with the driver using these changes is shown below.

```

test$ cat /sys/devices/virtual/fifo/fifo0/read_offset
0
test$ cat /sys/devices/virtual/fifo/fifo0/write_offset
0

test$ # (terminal 1)
test$ sudo ./fifow 10

test$ # (terminal 2)
test$ sudo ./fifow 20

test$ # (terminal 1)
test$ sudo ./fifow 11

test$ # (terminal 2)
test$ sudo ./fifow 21

test$ # (terminal 1)
test$ sudo ./fifor 2
20
21

test$ # (terminal 2)
test$ sudo ./fifor 2
10

test$ cat /sys/devices/virtual/fifo/fifo0/read_offset
0
test$ cat /sys/devices/virtual/fifo/fifo0/write_offset
0

```

It should be apparent that the driver no longer has any concurrency problems. Both terminals tried to write two values. Since the FIFO size is three one is discarded. When each terminal tries to read two values it can be seen that only three are obtained in total, which is correct. The order is difficult to predict but 21 coming after 20 is correct. And it is interesting that terminal 2 received value 10 which was written by terminal 1. And finally, it can be seen that the offsets are the same when it is empty after reading all the values out.

References

Corbet, J., A. Rubini, and Greg. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 2009. ISBN: 9780596555382.

Debian. *Debian – The Universal Operating System*. [Online; accessed 15-August-2013]. 2013. URL: <http://www.debian.org>.

Kroah-Hartman, Greg. *container_of()*. [Online; accessed 10-August-2013]. 2005. URL: http://www.kroah.com/log/linux/container_of.html.

Love, R. *Linux Kernel Development*. Developer's Library. Pearson Education, 2010. ISBN: 9780768696790.

Love, Robert. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, 2013. ISBN: 9781449341541.

Venkateswaran, S. *Essential Linux Device Drivers*. Pearson Education, 2008. ISBN: 9780132715812.