Learning Linux Device Drivers

Jeremiah Mahler (jmmahler@gmail.com)

August 16, 2013

Contents

1 Introduction **2**

2 Hello, World **3**
2.1 hello . . . 3
2.2 param . . . 4

3 Character Devices **6**
3.1 data_chr . . . 6
3.2 data_rw . . . 8
3.3 data_sk . . . 12
3.4 data_ioctl . . . 13
3.5 null, zero . . . 17
3.6 fifo_rw . . . 18

4 Sysfs **21**
4.1 fifo_sysfs . . . 21

5 Concurrency **24**
5.1 fifo_xxx . . . 24

1 Introduction

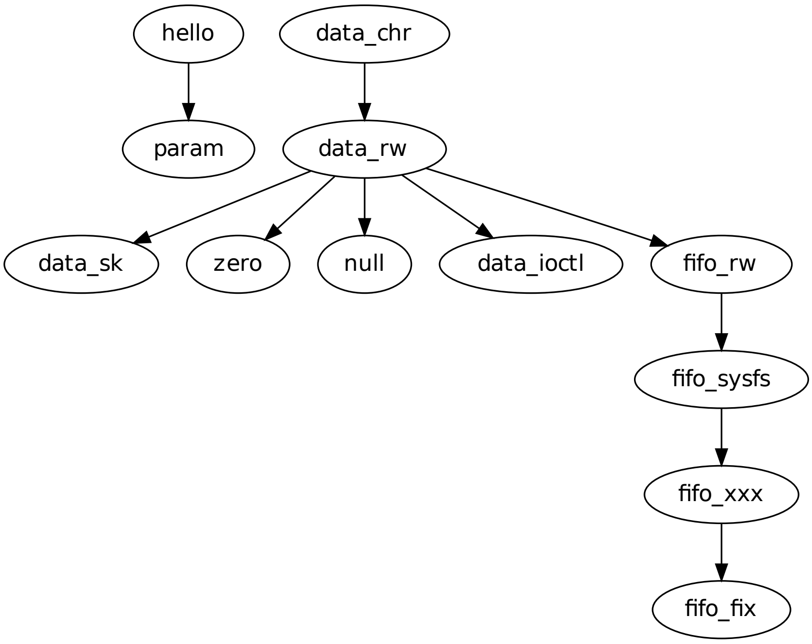

Figure 1: Hierarchy of kernel module examples. Simplest at the top downward to the more complex.

There are many excellent books about Linux device drivers[1][2][3].[4] However, in this authors experience, they were difficult to learn from. It certainly was not from their lack of detail. Clearly each of the authors have a profound understanding of the Linux kernel and their books reflect this. If anything it is due to this lack of simplicity.

The drivers described in this document aim to be simple and concise. Each one introduces as few concepts as possible. And each driver is a fully working example [5]. Many of the drivers are built in stages. Each stage introduces a new concept. And the changes are concisely described showing the differences (`diff`). Figure 1 shows the hierarchy of driver examples.

These examples were built using Kernel version 3.9 and 3.10. 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)
$
```

[1] J. Corbet, A. Rubini, and Greg. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 2009. ISBN: 9780596555382.
[2] S. Venkateswaran. *Essential Linux Device Drivers*. Pearson Education, 2008. ISBN: 9780132715812.
[3] R. Love. *Linux Kernel Development*. Developer's Library. Pearson Education, 2010. ISBN: 9780768696790.
[4] Robert. Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, 2013. ISBN: 9781449341541.
[5] Be creative with the examples. Try changing something and see what happens. Actively exploring in this way is a great way to solidify your understanding.

It may be necessary to install the kernel sources before compiling. To do this under a Debian[6] system the following steps can be used.

```
$ uname -r
3.9-1-amd64
$ apt-get install linux-source-3.9 linux-headers-3.9-1-amd64
```

2 Hello, World

2.1 hello

The `hello` module (Listing 1) simply prints message when it is loaded and unload.

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

```c
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.[7]

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

[7] Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 52.

The printk statements are the printf of the kernel domain. There are various 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.[8] There are various other MODULE_* options as well (linux/module.h).

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.

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 [9].

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

[8] Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 51.

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

```
--- ../hello/hello.c       2013-08-09 12:23:58.222416131 -0700
+++ hello.c 2013-08-09 12:53:38.082434726 -0700
@@ -1,15 +1,28 @@
 #include <linux/init.h>
 #include <linux/module.h>
+#include <linux/moduleparam.h>
+
+static int howmany = 1;
+module_param(howmany, int, S_IRUGO);

 static int __init hello_init(void)
 {
-    printk(KERN_ALERT "Hello, World\n");
+    int i;
+
+    for (i = 0; i < howmany; i++) {
+        printk(KERN_ALERT "Hello, World\n");
+    }
+
     return 0;
 }

 static void __exit hello_exit(void)
 {
-    printk(KERN_ALERT "Goodbye, cruel world\n");
+    int i;
+
+    for (i = 0; i < howmany; i++) {
+        printk(KERN_ALERT "Goodbye, cruel world\n");
+    }
 }

 MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
```

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

3 Character Devices

The `data` module allocates some memory which can then be read from and written to. This is accomplished as a character device and supports all the usual file operations.

3.1 data_chr

The first step is to construct the basic infrastructure for a character driver as shown in Listing 3. It doesn't do anything useful but it will simplify the description of upcoming drivers.

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 19-21) in this case only defines the `.owner`. Upcoming modules will add references to the open, close, read, write, and seek functions to this structure.

`alloc_chrdev_region` (line 26) allocates a major and minor number for the character device.[10] 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 left in a consistent state. A common way this is done is using `goto` statements which branch to different steps in the exit sequence.[11] 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 is also represented 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 it must be allocated before it is used (line 32).

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`.

[10] Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 66.
[11] Ibid., Pg. 53.

```c
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "data"

static dev_t data_major;
struct class *data_class;
struct device *data_device;

struct data_dev {
    struct cdev cdev;
} *data_devp;

struct file_operations data_fops = {
    .owner = THIS_MODULE,
};

static int __init data_init(void)
{
    int err = 0;

    err = alloc_chrdev_region(&data_major, 0, 1, DEVICE_NAME);
    if (err < 0) {
        printk(KERN_WARNING "Unable to register device\n");
        goto err_chrdev_region;
    }

    data_class = class_create(THIS_MODULE, DEVICE_NAME);

    data_devp = kmalloc(sizeof(struct data_dev), GFP_KERNEL);
    if (!data_devp) {
        printk(KERN_WARNING "Unable to kmalloc data_devp\n");
        err = -ENOMEM;
        goto err_malloc_data_devp;
    }

    cdev_init(&data_devp->cdev, &data_fops);
    data_devp->cdev.owner = THIS_MODULE;
    err = cdev_add(&data_devp->cdev, data_major, 1);
    if (err) {
        printk(KERN_WARNING "cdev_add failed\n");
        goto err_cdev_add;
    }

    data_device = device_create(data_class, NULL,
                        MKDEV(MAJOR(data_major), 0), NULL, "data%d",0);
    if (IS_ERR(data_device)) {
        printk(KERN_WARNING "device_create failed\n");
        err = PTR_ERR(data_device);
        goto err_device_create;
    }
```

```

    return 0;  /* success */

err_device_create:
    cdev_del(&data_devp->cdev);
err_cdev_add:
    kfree(data_devp);
err_malloc_data_devp:
    class_destroy(data_class);
    unregister_chrdev_region(data_major, 1);
err_chrdev_region:

    return err;
}

static void __exit data_exit(void)
{
    device_destroy(data_class, data_major);

    cdev_del(&data_devp->cdev);

    kfree(data_devp);

    class_destroy(data_class);

    unregister_chrdev_region(data_major, 1);
}

MODULE_AUTHOR("Jeremiah Mahler <jmmahler@gmail.com>");
MODULE_LICENSE("GPL");

module_init(data_init);
module_exit(data_exit);
```

Listing 3: data_chr driver.

3.2 data_rw

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 the maximum amount which in this case was 128 bytes. This maximum size is a `#define` inside the driver.

```
$ 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 compared to the previous `data_chr` driver. An array of bytes has been added to the per device structure along with the current offset (lines 7-17).

File operations for open, read, write and release have been added (lines 82-88). The release operation 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[12].[13] 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 functions sets the offset to zero (line 24) when 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 (lines 31-52) is called. Since the amount of data that can be read is limited by `MAX_DATA` the amount requested will be reduced it it is too large (lines 39-43). Then the `copy_to_user` function is used to attempt to transfer the data in to user space (lines 45-47). The `copy_to_user` also hecks to make sure that destination it is transferring to is valid for the given process. If the transfer was a success the new offset is stored (line 49) and then the number of bytes that were successfully transfered are returned (line 51).

The write function (lines 54-70) has the same operation as read except in the opposite directory. Notice that the `copy_from_user` (line 68) function is used in this case.

And since nothing needs to be done when the device is closed, the release function (lines 77-80) simply returns success.

---

[12] Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 79.

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

```
--- ../data_chr/data.c	2013-08-10 10:17:01.359016284 -0700
+++ data.c	2013-08-16 21:55:03.833816746 -0700
@@ -6,6 +6,7 @@
 #include <linux/uaccess.h>
 
 #define DEVICE_NAME "data"
+#define MAX_DATA 128
 
 static dev_t data_major;
 struct class *data_class;
@@ -13,10 +14,81 @@
 
 struct data_dev {
     struct cdev cdev;
+    char data[MAX_DATA];
+    loff_t cur_ofs;  // current offset
 } *data_devp;
 
+static int data_open(struct inode* inode, struct file* filp)
+{
+    struct data_dev *data_devp;
+
+    data_devp = container_of(inode->i_cdev, struct data_dev, cdev);
+    data_devp->cur_ofs = 0;
+
+    filp->private_data = data_devp;
+
+    return 0;
+}
+
+static ssize_t data_read(struct file *filp, char __user *buf,
+                              size_t count,
+                              loff_t *f_pos)
+{
+    struct data_dev *data_devp = filp->private_data;
+    loff_t cur_ofs;
+    char *datp;
+    size_t left;
+
+    cur_ofs = data_devp->cur_ofs;
+    datp = data_devp->data;
+    left = MAX_DATA - cur_ofs;
+
+    count = (count > left) ? left : count;
+
+    if (copy_to_user(buf, (void *) (datp + cur_ofs), count) != 0) {
+        return -EIO;
+    }
+
+    data_devp->cur_ofs = cur_ofs + count;
+
+    return count;
+}
+
+static ssize_t data_write(struct file *filp, const char __user *buf,
```

```
+                              size_t count,
+                              loff_t *f_pos)
+{
+   struct data_dev *data_devp = filp->private_data;
+   loff_t cur_ofs;
+   char *datp;
+   size_t left;
+
+   cur_ofs = data_devp->cur_ofs;
+   datp = data_devp->data;
+   left = MAX_DATA - cur_ofs;
+
+   count = (count > left) ? left : count;
+
+   if (copy_from_user((void *) (datp + cur_ofs), buf, count) != 0) {
+       return -EIO;
+   }
+
+   data_devp->cur_ofs = cur_ofs + count;
+
+   return count;
+}
+
+static int data_release(struct inode *inode, struct file *filp)
+{
+   return 0;
+}
+
 struct file_operations data_fops = {
    .owner = THIS_MODULE,
+   .open = data_open,
+   .read = data_read,
+   .write = data_write,
+   .release = data_release,
 };

 static int __init data_init(void)
@@ -35,7 +107,7 @@
    if (!data_devp) {
        printk(KERN_WARNING "Unable to kmalloc data_devp\n");
        err = -ENOMEM;
-       goto err_malloc_data_devp;
+       goto err_malloc_devp;
    }

    cdev_init(&data_devp->cdev, &data_fops);
@@ -60,7 +132,7 @@
    cdev_del(&data_devp->cdev);
 err_cdev_add:
    kfree(data_devp);
-err_malloc_data_devp:
+err_malloc_devp:
    class_destroy(data_class);
    unregister_chrdev_region(data_major, 1);
 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 requires the addition of one more function along with its corresponding entry in the file operations. Listing 5 shows the differences.

```
--- ../data_rw/data.c   2013-08-16 21:55:03.833816746 -0700
+++ data.c  2013-08-16 21:56:15.829814250 -0700
@@ -78,6 +78,35 @@
     return count;
 }
 
+static loff_t data_llseek(struct file *filp, loff_t offset, int orig)
+{
+    struct data_dev *data_devp = filp->private_data;
+    loff_t cur_ofs;
+
+    cur_ofs = data_devp->cur_ofs;
+
+    switch (orig) {
+        case SEEK_SET:
+            cur_ofs = offset;
+            break;
+        case SEEK_CUR:
+            cur_ofs += offset;
+            break;
+        case SEEK_END:
+            cur_ofs = MAX_DATA + offset;
+            break;
+        default:
+            return -EINVAL;
+    }
+
+    if (cur_ofs < 0 || cur_ofs >= MAX_DATA)
+        return -EINVAL;
+
+    data_devp->cur_ofs = cur_ofs;
+
+    return cur_ofs;
+}
+
 static int data_release(struct inode *inode, struct file *filp)
 {
     return 0;
@@ -88,6 +117,7 @@
     .open = data_open,
     .read = data_read,
     .write = data_write,
+    .llseek  = data_llseek,
     .release = data_release,
 };

```

```
@@ -107,7 +137,7 @@
     if (!data_devp) {
         printk(KERN_WARNING "Unable to kmalloc data_devp\n");
         err = -ENOMEM;
-        goto err_malloc_devp;
+        goto err_malloc_data_devp;
     }

     cdev_init(&data_devp->cdev, &data_fops);
@@ -132,7 +162,7 @@
     cdev_del(&data_devp->cdev);
 err_cdev_add:
     kfree(data_devp);
-err_malloc_devp:
+err_malloc_data_devp:
     class_destroy(data_class);
     unregister_chrdev_region(data_major, 1);
 err_chrdev_region:
```

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

To test the seek operations a seek-able cat program named `cats` has been created.

```
jeri@crowe:~/ldd/data_sk/test$ sudo wc -c /dev/data0
128 /dev/data0
jeri@crowe:~/ldd/data_sk/test$ sudo dd if=cats.c of=/dev/data0 bs=128 count=1
jeri@crowe:~/ldd/data_sk/test$ sudo ./cats /dev/data0 END -28
t the reset of 'file.txt'
 *jeri@crowe:~/ldd/data_sk/test$
jeri@crowe:~/ldd/data_sk/test$ sudo ./cats /dev/data0 SET 100
t the reset of 'file.txt'
 *jeri@crowe:~/ldd/data_sk/test$ exit
```

Notice that the file is 128 bytes total. The output after seeking from the start forward 100 bytes produces the same result as seeking backward 28 bytes from the end. In this instance the device is operating correctly.

3.4 data_ioctl

Using `ioctl()` for new designs is not recommended[14].[15] Instead `sysfs` is preferred. While `/proc` is another option, it is also becoming obsolete in favor of `sysfs`. Nonetheless it is still used so it is worth knowing how it works.

Include with this driver is a test program (`data_ioctl/test/ioctlx.c`). The driver allows the reset, read and write of a single global variable (x) in the driver. The test program allows these operations to be performed.

```
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$
```

[14] Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 156.
[15] Love, *Linux Kernel Development*, see n. 3.

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.[16] The user program must also contain the corresponding magic values as is done in the test program (Listing 7).

The three defines (lines 23-24) describe the supported ioctl operations. Any number of additional operations can be added. And it can be seen that there is an operation that has no data (`_IO`), reads data (`_IOR`) and writes data (`_IOW`). The `DATA_IOC_MAXNR` (line 27) is used as a sanity check later (line 41).

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

[16] Corbet, Rubini, and Kroah-Hartman, see n. 1, Pg. 158.

```
--- ../data_chr/data.c  2013-08-10 10:17:01.359016284 -0700
+++ data.c  2013-08-11 11:08:06.287053188 -0700
@@ -15,14 +15,92 @@
     struct cdev cdev;
  } *data_devp;

+
+int x;
+
+static int data_open(struct inode *inode, struct file *filp)
+{
+    struct data_dev *data_devp;
+
+    data_devp = container_of(inode->i_cdev, struct data_dev, cdev);
+
+    filp->private_data = data_devp;
+
+    return 0;
+}
+
+#define DATA_IOC_MAGIC 'm'
+
+#define DATA_IOCRESET _IO(DATA_IOC_MAGIC, 1)
+#define DATA_IOCWX _IOW(DATA_IOC_MAGIC, 2, int)
+#define DATA_IOCRX _IOR(DATA_IOC_MAGIC, 3, int)
+
+#define DATA_IOC_MAXNR 3
+
+static long data_ioctl(struct file *filp, unsigned int cmd,
+                unsigned long arg)
+{
+    int err = 0;
+    int retval = 0;
+
+    //struct data_dev *data_dev = filp->private_data;
+
+    if (_IOC_TYPE(cmd) != DATA_IOC_MAGIC) {
+        printk(KERN_ALERT "invalid ioctl magic\n");
+        return -ENOTTY;
+    }
+    if (_IOC_NR(cmd) > DATA_IOC_MAXNR) {
+        printk(KERN_ALERT "ioctl beyond maximum\n");
+        return -ENOTTY;
+    }
+
+    if (_IOC_DIR(cmd) & _IOC_READ)
+        err = !access_ok(VERIFY_WRITE, (void __user *) arg, _IOC_SIZE(cmd));
+    else if (_IOC_DIR(cmd) & _IOC_WRITE)
+        err = !access_ok(VERIFY_READ, (void __user *) arg, _IOC_SIZE(cmd));
+    if (err)
+        return -EFAULT;
+
+    switch (cmd) {
+        case DATA_IOCRESET:
+            /* takes no argument, sets values to default */
```

```
+               x = 0;
+               break;
+           case DATA_IOCRX:
+               /* read integer */
+               retval = __put_user(x, (int __user *) arg);
+               break;
+           case DATA_IOCWX:
+               /* write integer */
+               retval = __get_user(x, (int __user *) arg);
+               break;
+           default:
+               return -ENOTTY;   /* POSIX standard */
+               //return -EINVAL;   /* common */
+               /* Pg. 161 Linux Device Drivers (2005) */
+       }
+
+       return retval;
+}
+
+int data_release(struct inode *inode, struct file *filp)
+{
+       return 0;
+}
+
 struct file_operations data_fops = {
        .owner = THIS_MODULE,
+       .open = data_open,
+       .unlocked_ioctl = data_ioctl,
+       .release = data_release,
 };

 static int __init data_init(void)
 {
        int err = 0;

+       x = 0;
+
        err = alloc_chrdev_region(&data_major, 0, 1, DEVICE_NAME);
        if (err < 0) {
            printk(KERN_WARNING "Unable to register device\n");
```

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

```
#define DEVFILE "/dev/data0"

#define DATA_IOC_MAGIC  'm'

#define DATA_IOCRESET  _IO(DATA_IOC_MAGIC,  1)
#define DATA_IOCWX     _IOW(DATA_IOC_MAGIC, 2, int)
#define DATA_IOCRX     _IOR(DATA_IOC_MAGIC, 3, int)

int main(int argc, char* argv[])
{
    int devfd;
```

Listing 7: Corresponding "magic" in user program.

3.5 null, zero

From what has been described so far it is easy construct a driver for the well known /dev/null and /dev/zero devices.

The zero device is even simpler than the data_rw example (Section 3.2). The only real difference, other than names (data -> null), is the read and write operations as shown in Listing 8.

```
static ssize_t null_read(struct file *filp, char __user *buf,
                         size_t count, loff_t *f_pos)
{
    return 0;
}

static ssize_t null_write(struct file *filp, const char __user *buf,
                          size_t count, loff_t *f_pos)
{
    return count;
}
```

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.

```
static ssize_t zero_read(struct file *filp, char __user *buf,
                         size_t count, loff_t *f_pos)
{
    if (clear_user((void __user *) buf, count) > 0) {
        return -EFAULT;
    }

    return count;
}

ssize_t zero_write(struct file *filp, const char __user *buf,
                   size_t count, loff_t *f_pos)
{
    return count;
}
```

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 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 the fifo size is three (`#define MAX_DATA`, line 9). Once the fifo is full it cannot accept any more 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).

```c
    fifo_devp->fifo_start = &fifo_devp->fifo[0];
    fifo_devp->fifo_end = &fifo_devp->fifo[MAX_DATA-1];
    fifo_devp->read_ptr = &fifo_devp->fifo[0];
    fifo_devp->write_ptr = &fifo_devp->fifo[0];
    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 until there are no more left or it runs out of room. Since it can be both empty or full when the read pointer is at the same position as the write pointer an empty flag is used. Any successful write will make empty false (Listing 12, lines 88-89). Any successful read which results in the read pointer being the same as the write pointer will make empty true (Listing 11, lines 61-62).

```c
static ssize_t fifo_read(struct file *filp, char __user *buf,
                         size_t count,
                         loff_t *f_pos)
{
    struct fifo_dev *dev = filp->private_data;
    size_t left;

    left = count;

    while (left) {

        if (dev->empty) {
            break;
        }

        if (copy_to_user(buf, (void *) dev->read_ptr, 1) != 0) {
            return -EIO;
        }
        left--;

        if (dev->read_ptr == dev->fifo_end) {
            dev->read_ptr = dev->fifo_start;
        } else {
            (dev->read_ptr)++;
        }

        if (dev->read_ptr == dev->write_ptr) {
            dev->empty = 1;
        }
    }

    return (count - left);
}
```

Listing 11: fifo_rw driver read function.

```c
static ssize_t fifo_write(struct file *filp, const char __user *buf,
                          size_t count,
                          loff_t *f_pos)
{
    struct fifo_dev *dev = filp->private_data;
    size_t left;

    left = count;

    while (left) {

        if (!(dev->empty) && (dev->read_ptr == dev->write_ptr)) {
            break;
        }

        if (copy_from_user((void *) dev->write_ptr, buf, 1) != 0) {
            return -EIO;
        }
        left--;

        if (dev->empty)
            dev->empty = 0;

        if (dev->write_ptr == dev->fifo_end) {
            dev->write_ptr = dev->fifo_start;
        } else {
            (dev->write_ptr)++;
        }
    }

    return (count - left);
}
```

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 manner. But there was no way to get any info about the fifo from user space. Things such as is it empty/full and where are the read/write pointers at?

One way to create access to these metrics 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 added are the operations for reading and writing. With sysfs operations it is common to name them *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 132) will create a variable named `dev_attr_read_offset`.

There are many different kernel structures and attributes can be added to any of them. These other structures 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 the was created earlier is used along with `fifo_device` structure.

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 it is not the correct 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 that were created are there. There are also some extra attributes that were automatically created.

```
static ssize_t read_offset_show(struct device *dev,
                                struct device_attribute *attr,
                                char *buf)
{
    struct fifo_dev *fifo_devp = dev_get_drvdata(dev);
    return sprintf(buf, "%u\n", (unsigned int) (fifo_devp->read_ptr
                                                  - fifo_devp->fifo_start));
}

static ssize_t read_offset_store(struct device *dev,
                                    struct device_attribute *attr,
                                    const char *buf,
                                    size_t count)
{
    return 0;  // cannot store anything
}

static DEVICE_ATTR(read_offset, 0666, read_offset_show, read_offset_store);

static ssize_t write_offset_show(struct device *dev,
                                struct device_attribute *attr,
                                char *buf)
{
    struct fifo_dev *fifo_devp = dev_get_drvdata(dev);
    return sprintf(buf, "%u\n", (unsigned int) (fifo_devp->write_ptr
                                                  - fifo_devp->fifo_start));
}

static ssize_t write_offset_store(struct device *dev,
                                    struct device_attribute *attr,
                                    const char *buf,
                                    size_t count)
{
    return 0;  // cannot store anything
}

static DEVICE_ATTR(write_offset, 0666, write_offset_show, write_offset_store);
```

Listing 13: fifo_sysfs driver show/store.

```
    err = device_create_file(fifo_device, &dev_attr_read_offset);
    if (err) {
        goto err_file_read_offset;
    }

    err = device_create_file(fifo_device, &dev_attr_write_offset);
    if (err) {
        goto err_file_write_offset;
    }
```

Listing 14: fifo_sysfs driver attribute file creation.

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 concurrency issues. When multiple processes access a shared resource without restriction (such as with mutexes or semaphores) the resulting race conditions can wreak havoc. For example if a pointer offset was incremented beyond its maximum a write could easily crash the system.

Even if multiple processes access a resource the negative effects of race conditions may still be unlikely and difficult to reproduce. The technique used here is to first add code which makes the negative effects virtually guaranteed. And then it is shown how these problems can be fixed.

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 that point before allowing them to proceeding on. Listing 15 shows the additions.

```
+DEFINE_PWAIT(fifo_read);
+
 static ssize_t fifo_read(struct file *filp, char __user *buf,
                          size_t count,
                          loff_t *f_pos)
@@ -48,6 +82,8 @@
                 break;
         }

+        pwait_fifo_read();
+
         if (copy_to_user(buf, (void *) dev->read_ptr, 1) != 0) {
                 return -EIO;
         }
@@ -67,6 +103,8 @@
     return (count - left);
 }

+DEFINE_PWAIT(fifo_write);
+
 static ssize_t fifo_write(struct file *filp, const char __user *buf,
                           size_t count,
                           loff_t *f_pos)
@@ -78,6 +116,8 @@

     while (left) {

+        pwait_fifo_write();
+
         if (!(dev->empty) && (dev->read_ptr == dev->write_ptr)) {
                 break;
         }
```

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 indicate otherwise since they are unequal. 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 condtions.

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 and 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 creat race conditions in fifo_xxx driver.

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.