Project 2: Linux Kernel Modules

CS 311

Due May 4, 2012 at 12:00 pm, 60 pts.

For this project, you'll implement a fifo character device as a loadable kernel module. A fifo is a device which can store a finite number of characters, in our case 32 (defined by `FIFO_LEN` in `fifodev.h`), returning them in the order in which they were written. Of course, more than 32 characters can be written to the fifo, but it can only store 32 at a time. Note that reads "drain" the fifo.

The fifo you implement will have several processing modes, settable by invoking ioctls. There are two input modes:

1. `STOP`: Writes to the fifo will not overwrite unread data in the fifo.
2. `OVERWRITE`: Writes to the fifo will overwrite unread data.

Additionally, there are three character processing modes:

1. `WHISPER`: All uppercase letters should be converted to lowercase.
2. `NORMAL`: Perform no character processing.
3. `SHOUT`: All lowercase letters should be converted to uppercase.

The fifo will be readable and writable through ioctls as well as the normal filesystem interface.

Description

Refer to Section 7.1 of *The Linux Kernel Module Programming Guide* (`http://www.tldp.org/LDP/lkmpg/2.6/html/index.html`) for background. Requirements:

1. Only one process at a time should be able to use your fifo.
2. Use `fifodev.h` (see the course web site) as is, to ensure interoperability. Your module is expected to interoperate with `testFifo.c` (see the course web site) as is.
3. Your module should implement the following filesystem operations: read, write, ioctl, open, and release (close).
4. As described in `fifodev.h`, you should implement the following ioctls:
   (a) `IOCTL_SET_IMODE`, which takes `STOP` or `OVERWRITE` as a parameter and returns nothing.
   (b) `IOCTL_SET_PMODE`, which takes `WHISPER`, `NORMAL`, or `SHOUT` as a parameter and returns nothing.

(c) `IOCTL_WRITE_FIFO`, which takes a pointer to a `buffer_descriptor` as a parameter and returns the number of characters written to the fifo from the buffer. In `STOP` mode, the return value should be the actual number of characters written to the fifo. In `OVERWRITE` mode, the return value should be the length provided in the `buffer_descriptor`, assuming no errors occur. The current processing mode should be applied to all characters written to the fifo.

(d) `IOCTL_READ_FIFO`, which takes a pointer to a `buffer_descriptor` as a parameter and returns the number of characters read from the fifo to the buffer. Obviously, the return value can be no greater than `FIFO_LEN`.

5. Note that `buffer_descriptor` is a type defined in `fifodev.h` for interfacing with the two read/write ioctls. As described in `fifodev.h`, it is a struct consisting of the starting address of a character buffer and the buffer's length in characters.

6. Use the `#define` constants in `fifodev.h` wherever possible.

7. When the module is loaded, the fifo should be initialized so that it is empty and in `STOP` and `NORMAL` modes. Neither open nor close operations should affect fifo state, other than to ensure that at most one process can access the fifo at a time.

8. You will need to create the fifo's device file, `fifo_dev`. Refer to the comments at the head of `fifodev.h`.

9. Reading a `buffer_descriptor` from user address space to kernel address space will require using `copy_from_user()`. You would be wise to check the value returned by this function.

10. The fifo is in kernel address space, but the buffer is in user address space. Reading a character from the fifo to the buffer will require using `put_user()`. Writing a character from the buffer to the fifo will require using `get_user()`. You would be wise to check the values returned by these functions.

Testing

1. See `testFifo.c` for an incomplete fifo device test suite. You can refer to this for examples of how to invoke the `PMODE` and `IMODE` ioctls. I will be testing your modules against a stronger version of this test suite, so it would be to your advantage to think hard about what tests are missing and add them to the test suite I've given you. If you torture your module prior to turning it in, it's more likely to survive the torture to which I'm sure to subject it.

2. I would advise against attempting to implement all the fifo's functionality at once. I'd start with the four filesystem operations: open, release, read, and write, with the fifo in `NORMAL` and `STOP` modes. You can then test the fifo using the shell:

```
 % echo "aBc" > fifo_dev
 % echo "123 > fifo_dev
 % cat fifo_dev
aBc
123
 %
```

Note that `echo` appends a newline character. Therefore, each of the `echo`s above writes four characters to the fifo. If you don't want the trailing newline, use `echo -n`:

```
 % echo -n "aBc" > fifo_dev
 % echo "123 > fifo_dev
 % cat fifo_dev
aBc123
 %
```

Note that the `echo -n` above wrote just three characters to the fifo.

3. Once you get the fifo's basic functionality working, you can write five short programs to set each of the fifo modes and continue testing from the shell. For example, you can set `OVERWRITE` mode this way:

   ```
   int file = open(DEVICE_FILE_NAME, O_RDWR);
   ioctl(file, IOCTL_SET_IMODE, OVERWRITE);
   close(file);
   ```

   It would be wise to assert that `file >= 0` before using `file` following the call to `open()`.

   Once you've passed all your tests from the shell, move on to using `testFifo.c`.

4. I would use a judicious sprinkling of `printk()`s to monitor module operation, gating them with `#ifdef DEBUG` preprocessor directives:

   ```
   /* Uncomment the following #define to enable debugging messages. */
   /* #define DEBUG */

   ...

   #ifdef DEBUG
         printk(KERN_INFO, "Time for ice cream.\n");
   #endif
   ```

   Note that preprocessor directives should begin in the first column of a line, whereas the code affected by the directives is expected to be indented normally.

Project Turn-In

Email your module source file and any other relevant source file(s) to `kelliher[at]goucher.edu` by 12:00 pm on the 4th. Please note that this is a hard deadline.