Operating Systems
Lab 4
Linux Character Device Drivers

This lab takes a look at how device drivers are written and used. You will need to read chapters 3, 4, 7 and 9 of the Linux Kernel Module Programming Guide (called LKMPG) to get a better understanding of device drivers and to write the lab report. We will start with a very simple program and then build up to more sophisticated / complex device drivers. These lab exercises do not manipulate any real hardware, but will create pseudo character devices.

The first part of the lab uses code provided for you so that you can learn by example from reading the source code. The last two exercises require you to write a small amount of code. If you should not be successful at completing the last two parts of the lab, turn in as much as you have with documentation of what you were trying to do in the code and the nature of the problem that stopped you from being successful.

Procedure:

1. Use the wget command to download the tar file from the study guide.
   Extract the file using the command `tar xvf lab4.tar`.

2. Type the **`make`** command to compile chardev.c, and chario.c to loadable kernel module files.

3. The first module, chardev.c is discussed in chapter 4 of the LKMPG. One of the key points to understand about device drivers is that they present an abstraction of a file to the kernel. Notice the `file_operations` data structure which is registered with the kernel when the module is installed. The values in the data structure are pointers to functions contained in the device driver. Similar to the files in the /proc file system, which we studied in the last two labs, we need a real file which provides a handle to invoke the device driver and thus use the device. These *special* files are created with the `mknod` command. Read the man page for `mknod`. Answer questions 1 – 3.

4. Now install and use the chardev device driver. Normally, special device files are kept in the `/dev` directory, but since we are not

installing this driver as a regular part of the system, we will keep them under our home directory.

5. First, in another window, tail the system log file (`tail -f /var/log/messages`).
   `sudo /sbin/insmod ./chardev.ko`
   Now use `mknod` to make the special file. Look at the system log file for the command to use. Read the device driver file several times. (`cat hello`) Try to write to the device. First you will need to change the permissions of the special file to allow it to be written to. Observe the output from the system log file.
   ```
   sudo chmod go+w ~/labs/hello
   echo "hi" > hello
   sudo /sbin/rmmod chardev
   ```
   See question 4.

6. Next, we will look at chario.c, which is described in chapter 7 of the LKMPG. As before, install the module; make the special device file; write to the device and read from it. Don't remove the module just yet. See questions 5, 6, 7.
   `sudo /sbin/insmod ./chario.ko`
   See the log file for the options to the mknod command.
   ```
   sudo chmod go+w ~/labs/char_dev
   echo "hi" > char_dev
   cat char_dev
   ```

7. For devices which accept input and output, we need a special way to communicate with the device for control operations. We don't want the device to think that our control instructions are data being passed to the device. This is what the `ioctl() system call` is for. Some examples of how `ioctl()` is used are to eject a cdrom or set the baud rate of a modem. The `ioctl.c` file contains source code to be run as a user level process. Compile and run ioctl.c and observe the output on console and in system log file.
   ```
   gcc ioctl.c
   ./a.out
   ```
   Now remove the chario module.
   `sudo /sbin/rmmod chario`

8. The drivers looked at so far use the same simple mechanism to protect against two processes opening the device at the same time. They just return if the device is already open. A better approach is to have the second process block waiting for the device. (Recall the states that a process can be in.) Chapter 9 of the LKMPG discusses

how to block a process. Make a copy of the chario.c driver and modify the `device_open()` and `device_release()` functions according to the example in Chapter 9 of the LKMPG. In your lab report, be sure to show the code you wrote. Install your modified kernel module; make a character special device file; experiment with writing data to and reading data from the device. To see a process block and then be woken up, you may need to create a race condition. What I did was to write a user program that reads a file (like cat), but has a sleep statement to stall closing the file. (slow_read.c available on K-State Online) Note that you will also need to modify the Makefile to compile it. In your lab report, be sure to show the code you wrote.

```
gcc slow_read.c
cat Makefile > fifo
for i in 1
do
./a.out fifo &
cat Makefile > fifo &
done
```

9. The chario driver can be written to and read from, but it is not a true fifo (first in, first out) queue like a pipe is. Each read operation from a fifo queue should read new data if new data is available. The write operations should append data to the end of the current data in the queue. Make a copy of chario.c file, call it fifo2.c if you like. Modify the new driver to function as a true fifo queue. **Hint:** You will probably want to use an array to create a circular buffer. You will want to keep global pointers to where in the data the write operation last put data into the queue and where in the data the read operation last read from the queue.

10. Now remove any remaining modules you installed and delete the special device files which you created.

Questions:

1. What is the meaning of the parameters passed to mknod?

2. Study the source code of chardev.c and explain what the device does.

3. What is the purpose of the `try_module_get()` and `put_module()` macros? (You will need to do some reading to answer this.)

4. Why do we get the major number to use from the system log file? (Look at the source code to answer this.)

5. Study the source code of `chario.c` and describe what it does when written to and read from.

6. What do the `put_user()` and `get_user()` macros accomplish? Again, you may have to do a little reading to explain this.

7. Describe the mechanism used in `chario.c` to make sure only one process has the device open at any one time.

8. Study the source code for `ioctl.c` and the code in `chario.c` related to ioctl. How could the ideas in this code be used with real hardware?