CS 0449 – Project 4: /dev/rps

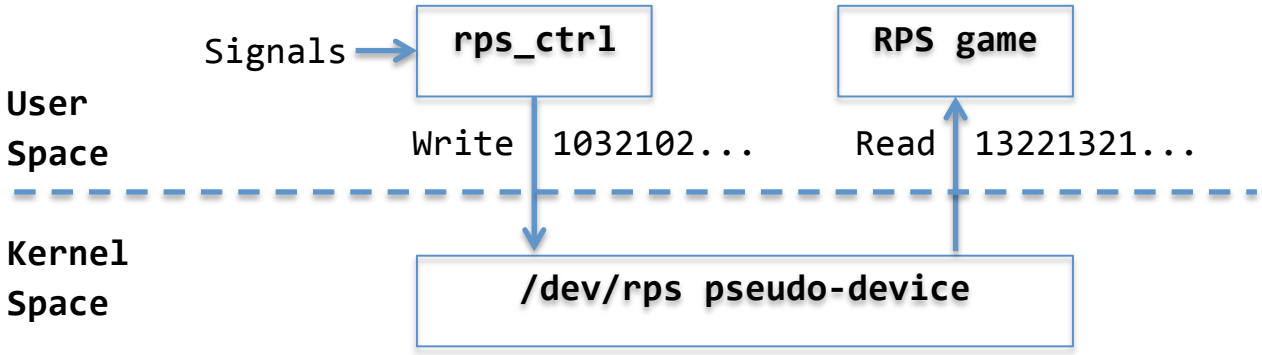
Due: Friday, December 8, 2017, at 11:59pm

Project Description

Standard UNIX and Linux systems come with a few special files like /dev/zero, which returns nothing but zeros when it is read, and /dev/urandom, which returns random bytes. These are device files that when read, invoke the device drivers underlying those files to provide the stream of bytes directed to the end user. These files are also called called pseudo-devices because there are no actual hardware devices that the device drivers control --- the device drivers emulate the given functionality in software. In this project, you will write a device driver for a pseudo-device named /dev/rps which, when read, returns a random stream of bytes consisting of 1s, 2s, and 3s. These numbers correspond to rock, paper, and scissors in the RPS game, respectively.

How It Will Work

Please review the Device Driver Lab before getting started. For this project we will need to create three parts: the /dev/rps pseudo-device, a rps_ctrl daemon program to control the pseudo-device, and your RPS game you wrote for Project 1 modified to use /dev/rps. At below is a schematic of the three components:



/dev/rps: This pseudo-device can be in one 4 modes: random mode, rock mode, paper mode, or scissors mode. Initially, it is in random mode by default. In random mode, it provides a stream of randomized 1s, 2s, and 3s when read. In rock mode, it provides a constant stream of 1s when read. In paper and scissors modes, it provides constant streams of 2s and 3s, respectively. The mode can be changed to random, rock, paper, or scissors by writing 0, 1, 2, or 3 to /dev/rps at any point in the operation of the device.

rps_ctrl: This is a daemon program that controls the /dev/rps pseudo-device. When it is launched, it opens the /dev/rps file and then immediately goes to sleep, and wakes up only when a signal is delivered to the program. It catches and handles 5 signals: SIGRTMIN, SIGRTMIN+1, SIGRTMIN+2, SIGRTMIN+3, and SIGTERM. The handlers for SIGRTMIN, SIGRTMIN+1, SIGRTMIN+2, and SIGRTMIN+3

write a single byte with the value 0, 1, 2, or 3 to the opened file /dev/rps respectively. The handler for SIGTERM closes /dev/rps and terminates the program.

RPS game: This is the RPS game you wrote for Project 1 modified to obtain random numbers by reading from /dev/rps, rather than calling rand(). The bytes read from /dev/rps can directly be translated to either rock, paper, or scissors. You should open /dev/rps once at the beginning of your program and close it at the end.

1. rps_ctrl Daemon Implementation

A daemon is a process that runs in the background that provide different services to the end user when requested. Programs such as mail servers and file transfer servers typically run as daemons. To run rps_ctrl as a daemon, you just have to include an & (ampersand) at the end of the command line:

```
thoth $ ./rps_ctrl &
thoth $
```

As seen above, your command prompt will return immediately after running rps_ctrl &, perhaps leading you to think that the program has terminated. But the program is still running in the background, as can be seen with ps:

```
thoth $ ps
  PID TTY          TIME CMD
10333 pts/4    00:00:00 bash
18976 pts/4    00:00:00 rps_ctrl
18977 pts/4    00:00:00 ps
thoth $
```

Since rps_ctrl runs in the background and is not interactive, there is no way for you to do Ctrl+C on the program to terminate it. But we learned in class we can used the kill utility to send a SIGTERM signal, which achieves the same thing:

```
thoth $ kill 18976
```

You can also send other signals to change device modes using kill:

```
thoth $ kill –SIGRTMIN 18976
```

Procedure

1. On thoth.cs.pitt.edu, in /u/SysLab/USERNAME copy over the skeleton code and compile:

   ```
   cp ~wahn/public/cs449/rps_ctrl.c ./
   gcc –m32 ./rps_ctrl.c –o ./rps_ctrl
   ```

2. The above generates a 32-bit version of rps_ctrl which can be run on our virtual machine. Launch your Linux virtual machine in VirtualBox, login using root/root and do the following:

   ```
   scp USERNAME@thoth.cs.pitt.edu:/u/SysLab/USERNAME/rps_ctrl .
   ```

3. Try launching and terminating the daemon on the virtual machine:

```
./rps_ctrl &
ps
kill <pid of rps_ctrl>
```

Please NEVER launch the daemon on thoth. Let's try to keep our shared machine free of clutter.

4. Modify rps_ctrl.c so that it does correct signal handling. For now, the skeleton code just does an infinite loop, waiting for signals. One difference with the SIGALRM handler program on the lecture slides is that the loop does a pause() at every iteration. Pause puts the daemon to sleep while waiting for a signal, preventing it from wasting valuable CPU cycles in a fruitless loop.

5. Make sure you use system calls such as open(), write(), and close() to write to /dev/rps. If you use C Library calls such as fopen(), fwrite(), and fclose(), the writes will get buffered in the internal buffers of the library and will not get sent to /dev/rps immediately. Refer to the Hello World example at the end of the system call slides. You only need to pass O_WRONLY to open() since the device file is already created.

6. After completing the next step, please add rps_ctrl.c to the rps_dev/ directory below so that you have a unified project directory. Also, please modify the Makefile in rps_dev to add another rule for rps_ctrl so that 'make' compiles both rps_ctrl and the device driver.

2. /dev/rps Device Driver Implementation

Our device driver will be a character device that will implement a read function (which is the implementation of the read() syscall) and returns an appropriate result (a 1 byte value from 1 to 3). As we discussed in class, the kernel does not have the full C Standard library available to it and so we need to get use a different function to get random numbers. By including <linux/random.h> we can use the function get_random_bytes(), which you can turn into a helper function to get a single byte:

```
unsigned char get_random_byte(int max, int min) {
        unsigned char c;
        get_random_bytes(&c, 1);
        return c%max + min;
}
```

Procedure

7. On thoth.cs.pitt.edu, in /u/SysLab/USERNAME do the following. Unlike the Device Driver Lab, we are able to compile on thoth now, since I installed the Debian kernel there:

```
tar xvfz ~wahn/public/cs449/rps_dev.tar.gz
cd rps_dev
make ARCH=i386
```

8. The above generates the rps_dev.ko device driver kernel object. Launch your Debian Linux virtual machine in VirtualBox, login using root/root and do the following:

```
scp USERNAME@thoth.cs.pitt.edu:/u/SysLab/USERNAME/rps_dev/rps_dev.ko .
```

9. Load the device driver using insmod:

```
insmod rps_dev.ko
```

10. When the device driver loads it prints a message on how to run mknod. Follow instructions to create the device file that will act as the interface to this device driver.

```
mknod /dev/rps c <MAJOR_NUMBER> <MINOR_NUMBER>
```

11. Now test the device file by running the following commands:

```
cat /dev/rps
echo "Hello" > /dev/rps
```

    For now, the device driver will just print some debug messages on system call parameters. Your job is to make the device driver actually work.

12. If you need to update the device driver, remove the module first before inserting it again:

```
rmmod rps_dev.ko
```

3. RPS Game Modification

Copy rps.c over to the rps_dev/ directory so that you have a unified project directory like we did for rps_ctrl. Also, modify the Makefile to add the rule for rps, so now 'make' produces three items: rps, rps_ctrl, and the device driver.

The only modification required is to read the random numbers from /dev/rps rather than relying on rand(). In this case, you are free to use C Library functions to perform file I/O since buffering is not an issue in this case (it is okay for numbers to be buffered for a while before they are consumed). If /dev/rps is in rock / scissors / paper mode, the computer player should always choose rock / scissors / paper respectively. Again, since the virtual machine is 32-bits, you should add –m32 when compiling:

```
gcc –m32 rps.c –o rps
```

Copy rps over to the virtual machine for testing, as we did before. We cannot run rps on `thoth.cs.pitt.edu` because its kernel does not have the device driver loaded. However, we can test the program on the virtual machine once we have installed the new driver.

File Backups

One of the major contributions the university provides for the AFS filesystem is nightly backups. However, the `/u/SysLab/` partition is **not** part of AFS space. Thus, any files you modify under your personal directory in `/u/SysLab/` are not backed up. If there is a catastrophic disk failure, all of your work will be irrecoverably lost. As such, it is my recommendation that you:

Backup all the files you change under `/u/SysLab` to your `~/private/` directory frequently!

Loss of work not backed up is not grounds for an extension. YOU HAVE BEEN WARNED.

Hints and Notes

- `printk()` is the version of `printf()` you can use for debugging messages from the kernel.
- In the driver, you can use some standard C functions, but not all. They must be part of the kernel to work.
- As root, typing `poweroff` in the virtual machine will shut it down cleanly.
- If the module crashes, it may become impossible to delete the file you created with mknod in /dev. If that happens, just grab a new disk image and start over. It's why we're developing in a virtual machine as opposed to a real one.
- You will be dealing with binary numbers (0, 1, 2, and 3) that are not printable characters. Use `xxd` to view a stream of binary numbers like we did in the Device Driver Lab. Also use `head` to truncate infinite streams and | pipes as appropriate.

Requirements and Submission

You need to submit rps_dev directory compressed into a tar.gz fie. Inside should be:

- Your `Makefile` that is able to compile rps, rps_ctrl, and rps_dev.ko on 'make'
- Your rps.c, rps_ctrl.c, and rps_dev.c source files, as well as any other required source files
- Please do 'make clean' before submission and not include any intermediate files

Make a tar.gz file named `USERNAME-project4.tar.gz`

Copy it to `~wahn/submit/449/RECITATION_CLASS_NUMBER` by the deadline for credit.