Homework #5: Driversity

Issued: Thursday, November 13
Due: Thursday, December 11

Purpose

This assignment asks you to develop a Linux module that implements a device driver. It should only be performed on a computer, or virtual machine (VM), intended for operating-system experiments. You have been warned!

> *So, you want to write a kernel module. You know C, you've written a few normal programs to run as processes, and now you want to get to where the real action is, to where a single wild pointer can wipe out your file system and a core dump means a reboot.*

Your driver will provide a software abstraction called a scanner. A *scanner* helps an application split a sequence of characters into a sequence of *tokens*, based on the positions of *separators*. For this assignment, a separator is a single character from a set of characters (e.g., space, tab, newline, and colon).

Resources

Our textbook discusses much of this material, in Chapter 36. In addition, a much more specific, and far more humorous treatment, is given in:

```
https://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf
https://tldp.org/LDP/lkmpg/2.6/html/index.html
```

The ominous quote, above, is from this most excellent document. Suggestions for protection are made below.

Example

Here's is an example application that uses such a scanner, on a file like `/etc/passwd`:

`pub/hw5/TryScanner.c`

You may be familiar with the `strtok()` scanner. For this assignment, you must develop your own scanner.

Skeleton

A rudimentary module/driver is provided. However, it lacks an implementation of many of the features required by this assignment. Start with this:

`pub/hw5/Hello`

Configuring a Development Environment

To avoid the disasters forewarned in the quote above, you do not want to test a driver on a production computer (i.e., one you care much about). One way to do this is to test your untrusted driver on a disposable VM:

```
1  onyxnode$ cd ~/tmp
2  onyxnode$ cp ~jbuffenb/classes/452/pub/hw5/fedora.img .
3  onyxnode$ ~jbuffenb/classes/452/pub/bin/hw5boot
4  onyxnode$ scp -P 2222 -r \
5            ~jbuffenb/classes/452/pub/hw5/Hello \
6            cs452@localhost:
7  onyxnode$ ssh -p 2222 cs452@localhost
8  [cs452@qemu ~]$ cd Hello
9  [cs452@qemu Hello]$ make
10 [cs452@qemu Hello]$ make install
11 [cs452@qemu Hello]$ make try
12 [cs452@qemu Hello]$ exit
13 onyxnode$ ssh -p 2222 cs452@localhost \
14           sudo -S shutdown -h 0
```

You can log into, and out of, your VM, as you wish, without shutting it down. However, you can only use your VM from the same node from which you started

it (with `hw5boot`). Please remember to `shutdown` your VM, when you are done with it; it will retain your work. Please remember to remove your `.img` file, when you are *really* done with it, since it is quite large.

There are two small tools, in our `pub/bin` directory, which you may find helpful. You can run `sshnode`, on `onyx`, to `ssh` to a randomly chosen node. You can run `log`, on your VM (in another window), to show the `printk()` debugging output produced by your driver.

If you want to install tools (e.g., `gdb`) on your VM, simply:

```
1 [cs452@qemu ~]$ sudo dnf install gdb
```

Requirements and Suggestions

1. You must develop a kernel module implementing a character driver for a scanner. However, I *strongly* suggest that you develop your "algorithmic" code in the much friendlier environment of a normal user-space program (e.g., with `gdb`). Kernel-space debugging is best avoided, when possible.

2. The `init()` function must establish a reasonable set of default separators. The default set can be overridden in particular scanner instances.

3. The `open()` function must create an instance of a scanner. A process can have multiple instances open concurrently. They might be scanning different data with different separators.

4. The set of separators is specified by calling `ioctl()` with a request of `0`, followed by calling `write()` with the separator characters.

5. The sequence of characters to scan is specified by calling `write()`. Such calls are not cumulative: each call specifies a new sequence to scan.

6. The next token is scanned by calling `read()`. It returns the number of characters scanned. If the length of the token exceeds the number of characters requested, another `read()` scans more of the same token.

   - A return value of `0` indicates "end of token."
   - A return value of `-1` indicates "end of data."

7. Your driver must not be biased toward C strings, or any other string representation. In particular, it must not treat the ASCII character NUL (i.e., `0x00`) specially. NUL may or may not be a separator. NUL may or may not be in the sequence of characters to scan. The separators and sequence may or may not end with a NUL.

8. You must develop and demonstrate a thorough test suite. The semantics of `read()` are difficult to implement correctly! If you do not demonstrate correctness, I'll assume the opposite.
9. You must provide good documentation and error messages.
10. You must avoid memory leaks, especially kernel memory leaks.
11. Your submission will be evaluated on `onyx`.