Assignment 1: Exploring the kernel through modules

CS614: Linux Kernel Programming

January 27, 2023

In this assignment, you have to implement character device drivers (one driver for each part). Name of the char device must be `cs614_device` (i.e. its absolute path will be `/dev /cs614_device`). Your driver should also create sysfs directory named `cs614_sysfs` within `/sys/kernel` directory (i.e. `/sys/kernel/cs614_sysfs`) and a file named `cs614_value` within this `cs614_sysfs` directory (i.e. `/sys/kernel/cs614_sysfs/cs614_value`).

User programs will communicate with your driver in following manner:

- User program will write some operation number (Example: 0, 1, 2 etc., also referred to as "command" in this document) to the sysfs file `/sys/kernel/cs614_sysfs/cs614_value`
- Your driver should do the specified operation mentioned below (under different parts of the assignment).
- User program will read the result through device file `/dev/cs614_device` using the `read` system call.

Example: If the user process writes '6' to the sysfs file `/sys/kernel/cs614_sysfs/cs614_value` to know the number of files opened by the process, the module will store this command. When the user program does `read()` operation on device file `/dev/cs614_device`, it should fill up the buffer argument of the `read` system call with th e number of files opened by the calling process. *Note:* The process setting the command through the sysfs file will read the result.

Assumptions

- The command set by the sysfs write is applied only at the time of handling the `read` on the chardev.
- In the testing, we will only user `open(), write(), close()` functionalities on the sysfs file `/sys/kernel/cs614_sysfs/cs614_value`
- No operation except `open(), read(), close()` will be used during testing for the device file `/dev/cs614_device`
- sysfs file `/sys/kernel/cs614_sysfs/cs614_value` will be created with 0660 permissions
- For values beyond '7', write for the sysfs file `/sys/kernel/cs614_sysfs/cs614_value` must return `-EINVAL`.
- Read operation on `/dev/cs614_device` will be done only after a write operation has happened on `/sys/kernel/cs614_sysfs/cs614_value`

Part1: Single Process Access [20 Marks]

In this question, you can assume that there will be only a single process that will use the character driver at any time. You have to implement support for following functionality in the character driver:

1. **Pid (command = 0):** Whenever value ‘0’ is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process and `read(fd, buf, size)` system call is made on the device file `/dev/cs614_device` by that user process, then, value of *pid (process id) of the user process* should be filled in the `buf` passed as argument to `read()` system call. Your `read()` system call should return the number of bytes read.

2. **Static priority (command = 1):** Whenever value ’1’ is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process and `read(fd, buf, size)` system call is made on the device file `/dev/cs614_device` by that user process, then, value of *static priority of the user process* should be filled in the `buf` passed as argument to `read()` system call. Your `read()` system call should return the number of bytes read.

3. **Command Name (command = 2):** Whenever value ‘2’ is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process and `read(fd, buf, size)` system call is made on the device file `/dev/cs614_device` by that user process, then, *name of the of the user process* should be filled in the `buf` passed as argument to the `read()` system call. Your `read()` system call should return value ‘n’ (where ‘n’ is the number of characters in the command name excluding the null terminator) in this case.

4. **Ppid (command = 3):** Whenever value ‘3’ is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process and `read(fd, buf, size)` system call is made on the device file `/dev/cs614_device` by that user process, then, value of *the real parent’s process id of the user process* should be filled in the `buf` passed as argument to `read()` system call. Your `read()` system call should return the number of bytes read.

5. **Number of voluntary context switches (command = 4):** Whenever value ‘4’ is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process and `read(fd, buf, size)` system call is made on the device file `/dev/cs614_device` by that user process, then, value of `number of voluntary context switches) of the user process` should be filled in the `buf` passed as argument to `read()` system call. Your `read()` system call should return value the number of bytes read.

Part2: Multiprocess Access [30 Marks]

In this part, you have to modify your kernel module (to implement an extended module) written in Part 1 to add the ability such that multiple processes can use your module simultaneously. For example: Consider that there are two processes $P1$ and $P2$ using the character driver. Inter-leavings like the following should give correct results:

$P1$ performs write on the sysfs file $command = 0$

$P2$ performs write on the sysfs file $command = 3$

$P1$ reads the chardev with a buf argument. Should return the PID of $P1$

$P2$ reads the chardev with a buf argument. Should return the PPID of $P2$

Note: Your implementation should not impose any limits on # of processes using the sysfs file and chardev.

Part3: Multithreaded Access [50 Marks]

In this part, you have to modify your kernel module written in part2 to support following functionalities. *Note that*, any thread in the process can use the following commands and any other thread in the process can read the result values using the chardev.

1. **Number of Threads (command = 5):** Whenever value ‘5’ is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process and `read(fd, buf, size)` system call

is made on the device file `/dev/cs614_device` by the user process/threads, then, value of *number of threads created by the user process* using `pthread_create()` + 1 (main/process thread itself) should be filled in the `buf` passed as argument to `read()` system call. Your `read()` system call should return the number of bytes read.

2. **Number of Open Files (command = 6):** Whenever value '6' is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process/thread and `read(fd, buf, size)` system call is made on the device file `/dev/cs614_device` by that user process, then, value of **number of files opened by the process/user thread** should be filled in the `buf` passed as argument to `read()` system call. Your `read()` system call should return the number of bytes read.

3. **Max Stack Usage PID (command = 7):** Whenever value '7' is written to `/sys/kernel/cs614_sysfs/cs614_value` by a user process and `read(fd, buf, size)` system call is made on the device file `/dev/cs614_device` by that user process or any of its threads, then *pid of the thread that consumes maximum amount of user stack among all threads including the main process* should be filled in the `buf` passed as argument to `read()` system call. Your `read()` system call should return the number of bytes read.

Organization and Testing

```
A1_release|
          |-CS614_Assignment1_2023.pdf
          |-examples
          |       |-chdev
          |       |-kernth
          |       |-traphook-sysfs
          |-Part1
          |      |-driver.c
          |      |-Makefile
          |      |-user_progs
          |-Part2
          |      |-driver.c
          |      |-Makefile
          |      |-user_progs
          |-Part3
          |      |-driver.c
          |      |-Makefile
          |      |-user_progs
```

The `examples` directory contains modules used in the class. Please refer to the class Piazza page. `Part1` to `Part3` contain a template file for your implementation along with the user test programs. To test your implementation, follow the steps mentioned below,

- The directory `user_progs` in each part contains open test cases for that part. You may design additional test cases for your implementation.

- Please refer to the `readme` file in each `user_progs` directory to know how to compile,run and use the test cases.

Submission

You have to submit *a single zip file* named **your_roll_number.zip**. For example if your roll no is 1211405, you should create the zip file named 1211405.zip containing **only** the following files in specified folder format. Note that, the zip file should expand into a directory named as your roll number.

```
RollNo|
      |-Part1
      |      |-driver.c
      |      |-Makefile
      |-Part2
      |      |-driver.c
      |      |-Makefile
      |-Part3
      |      |-driver.c
      |      |-Makefile
```