Kernel-Based Message Queue

1 Introduction

The objective of this assignment is to implement a kernel-based first-in-first-out (FIFO) message queue by extending a barebone character device driver. This message queue can be used by multiple producers / consumers to exchange data. With this assignment you will familiarize yourselves with concurrency primitives for mutexes. Specifically, you will:

- Extend a character device driver, implemented as a kernel module, to implement a FIFO message queue that supports multiple producers / consumers.
- Extend a user space program to interact with the character device driver to test it.

Important: put comments everywhere in your code to explain to the CAs what your code is doing!

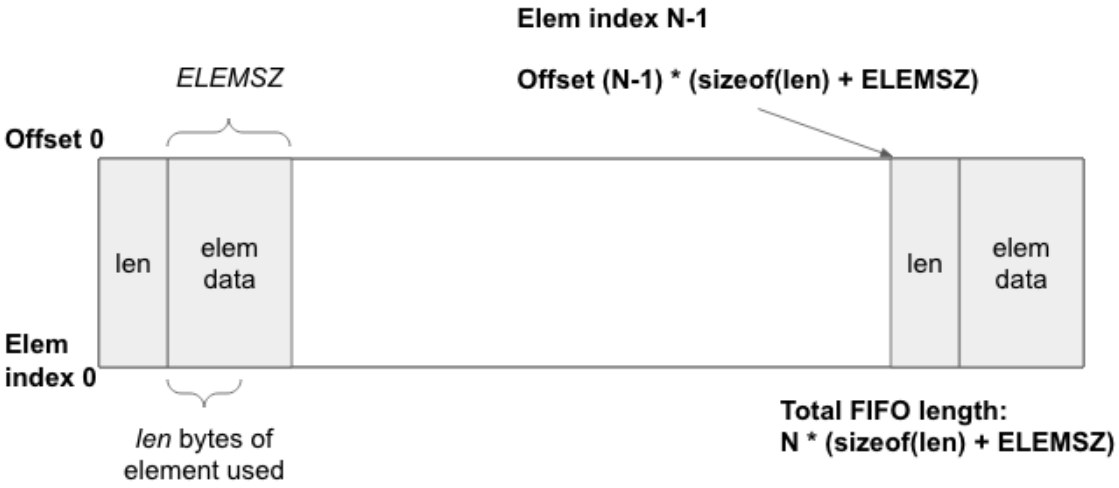
2 Installing and Testing the Given "scull" Character Device Driver

1) Boot your Debian virtual machine using your "known good" Linux kernel (most likely the one with a name ending with "amd64").
2) Download the file "pmeunier-pa5.zip" from Canvas to the Debian VM, unzip this archive file in your home directory, and rename the new "pmeunier-pa5" directory to use your own student login name instead of "pmeunier". Then go into this directory.
3) In the "scull" character device driver (under driver/) we have prepared one IOCTL command, which you can find in the **`scull_ioctl`** function of the "scull.c" file. This command simply returns the size of one element of the queue (in other words, the maximum length of any message exchanged between a producer and consumer), although the queue itself is not implemented yet (this will be your job below). The "scull.c" file also contains two **`scull_read`** and **`scull_write`** functions which currently do nothing except write an informational message to the kernel log. To test the code:
   a) Modify MODULE_AUTHOR in "scull.c" to indicate your own student login name.
   b) Compile the driver code (it is implemented as a kernel module).
   c) Load the module into the kernel. Also check what happens at the same time in the kernel log.
   d) Create the special character device file /dev/scull with the correct major number for the "scull" module (see Programming Assignment 4 if you do not remember how to do this).
   e) Since you will need to implement **`scull_read`** and **`scull_write`** operations for the message queue, you must also execute the following command to ensure that you ("**`o`**" means "other", which, from the point of view of the "root" administrator who is the owner of the /dev/scull file, includes you) have the proper read ("**`r`**") and write ("**`w`**") permissions for the /dev/scull file:
   `sudo chmod o+rw /dev/scull`
   f) Compile the user-space producer and consumer programs (under src/).
   g) Test the consumer and producer programs to get a feel of what they do. Also check what happens at the same time in the kernel log.
   h) Unload the module.

3 Implementing the Message Queue in The Character Device Driver

Extend the "scull" character device driver (under driver/) by implementing a FIFO message queue that can store up to N messages (N is called **`scull_fifo_size`** in the code), with each message being up to ELEMSZ characters long (called **`scull_fifo_elemsz`** in the code). The provided code is written in such a way that N and ELEMSZ can be configured during module load through module parameters (see below) so do not make any assumption in your own code about the values of N and ELEMSZ, except that you can assume that they are always positive integers. The only limits for N and ELEMSZ should be kernel memory availability, not anything in your implementation. Do NOT delete existing code and do NOT create new files. Only modify existing files to introduce your functionality. Your tasks:

1) Allocate the message queue as a flat array of characters (not as a list, not as an array of structures, not as multiple arrays; this is purely to make you practice pointer arithmetic) when the module is loaded into the kernel and correctly free the message queue when the module is unloaded. Information on memory allocation / deallocation in the kernel can be found here https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html
   a) You will need to check whether your memory allocation succeeds or not; if not, you should return an error.
   b) Because processes can store less than ELEMSZ characters of data in a given message, you will also need to always store with each queue element the number of characters actually stored in the element. See the picture below, where for example the length "len" for the first message is less than ELEMSZ, and so the "len" needs to always be stored as part of the array, in front of the data for the message proper.



   c) Since you need to use a flat array to implement a FIFO message queue, you will also need two pointers (not any integer type) to manage this array as a "circular buffer":
      1) **`start`** to indicate the oldest full array element where the next (oldest) message can be read;
      2) **`end`** to indicate the oldest empty array element where the next (newest) message can be written.
      3) Look at https://en.wikipedia.org/wiki/Circular_buffer to learn what a circular buffer is and how to simulate a queue using a flat array and two pointers (**`start`** and **`end`**).
      4) Make sure these two pointers are properly initialized when the module is loaded!

2) Implement the **`scull_read`** (consume) message queue operation in the device driver. A placeholder function is already in driver/scull.c that currently does nothing except print a message in the kernel log. This queue operation consumes one message stored in one queue element, copying the data available in the next full element into the **`buf`** function argument and returning the number of bytes copied as result, which obviously cannot be bigger than the amount of data available in the next full element. If **`count`** is smaller than the amount of data available in the next full element in the queue, then only **`count`** characters of the message are copied into **`buf`**. **`f_pos`** is not used. The **`scull_read`** operation must block if there is no message in the array to consume. It should return an error if the copying fails.
   Because the size ELEMSZ of the queue elements is not known at compile time, you will need to do pointer arithmetic when updating the **`start`** pointer after a read. Also make sure that **`start`** properly wraps around at the end of the array back to the beginning, as necessary, to correctly implement a circular buffer.
3) Implement the **`scull_write`** (produce) message queue operation in the device driver. A placeholder function is already in driver/scull.c that currently does nothing except print a message in the kernel log. This queue operation produces one message, copying **`count`** bytes from **`buf`** into the next empty queue element and returning the number of bytes copied as result, which cannot be bigger than ELEMSZ. If **`count`** is larger than ELEMSZ, then only ELEMSZ bytes are copied, otherwise **`count`** bytes are copied. **`f_pos`** is not used. The **`scull_write`** operation must block if there is no space in the array to write a new message. It should return an error if the copying fails.
   Because the size ELEMSZ of the FIFO elements is not known at compile time, you will need to do pointer arithmetic when updating the **`end`** pointer after a write. Also make sure that **`end`** properly wraps around at the end of the array back to the beginning, as necessary, to correctly implement a circular buffer.
4) Add synchronization code to **`scull_read`** and **`scull_write`** to ensure safe concurrent access to the message queue, as we saw in class in the "Concurrency Part 2" lecture notes. Use sleeping / waking synchronization functions, not busy-waiting ones.
   If you are using semaphores and mutexes functions to block the consumer and producer processes then use their interruptible variants (both for semaphores and mutexes), so that the functions can be interrupted if you decide to terminate the processes early using a signal (e.g., through Ctrl-c from the keyboard). Make sure your code checks and returns an appropriate error when that happens (and does not leave any semaphore or mutex still locked)!

4 Modifying the User-Space Test Programs

1) Modify the code of the producer (in src/) to use your own name as the message rather than "Philippe Meunier" (nothing else). Note how the string is written to the message queue without the trailing '\0' character, just for extra fun.
2) In the consumer code, you will notice that the consumer currently reads only one character from each queue element. Modify the code of the consumer so that it reads (and prints) the complete data from a queue element. Do not assume that the consumer program knows in advance the size of each queue element: the driver already implements an IOCTL command (SCULL_IOCGETELEMSZ) to return the size of a queue element (ELEMSZ) to user-space processes and this is what your code must use to learn the maximum size allowed for a message. So use this IOCTL command in the consumer code to dynamically allocate a user-space buffer of the right size before reading a queue element. Make sure

that a '\0' exists at the end of the data in the buffer before printing the data. Also make sure the user-space buffer is properly deallocated when it is not needed anymore.

5 Testing the Whole Message Queue

1) Load the module with a large **scull_fifo_size** and a small **scull_fifo_elemsz**. Start 2 producers and then 2 consumers. For example:

```
$ sudo insmod ../driver/scull.ko scull_fifo_size=10 scull_fifo_elemsz=3
$ ./producer p 2
Device (/dev/scull) opened
write: Philippe Meunier
write: Philippe Meunier
Device (/dev/scull) closed
$ ./consumer p 2
Device (/dev/scull) opened
read: Phi
read: Phi
Device (/dev/scull) closed
```

   You must provide a single screenshot that shows a Debian terminal window that shows output similar to what is shown above (including the **insmod** command used), followed by the output of the **id** command.

2) Reload the module with a large **scull_fifo_size** and a large **scull_fifo_elemsz**. Start 2 producers and then 2 consumers. For example:

```
$ sudo insmod ../driver/scull.ko scull_fifo_size=10 scull_fifo_elemsz=30
$ ./producer p 2
Device (/dev/scull) opened
write: Philippe Meunier
write: Philippe Meunier
Device (/dev/scull) closed
$ ./consumer p 2
Device (/dev/scull) opened
read: Philippe Meunier
read: Philippe Meunier
Device (/dev/scull) closed
```

   You must provide a single screenshot that shows a Debian terminal window that shows output similar to what is shown above (including the **insmod** command used), followed by the output of the **id** command.

3) Reload the module with a small **scull_fifo_size** and a large **scull_fifo_elemsz**. Start 5 producers and show that some of them block waiting for consumers (then use Ctrl-c to kill the remaining producers). Then start 5 consumers and show that some of them print messages and the remaining consumers block waiting for producers (then use Ctrl-c to kill the remaining consumers). For example:

```
$ sudo insmod ../driver/scull.ko scull_fifo_size=2 scull_fifo_elemsz=30
$ ./producer p 5
Device (/dev/scull) opened
write: Philippe Meunier
write: Philippe Meunier
write: Philippe Meunier
write: Philippe Meunier
```

```
write: Philippe Meunier
^C
$ ./consumer p 5
Device (/dev/scull) opened
read: Philippe Meunier
read: Philippe Meunier
^C
```

You must provide a single screenshot that shows a Debian terminal window that shows output similar to what is shown above (including the **insmod** command used), followed by the output of the **id** command.

4) Open two Debian terminal windows at the same time. In the first terminal, reload the module with the same small **scull_fifo_size** and the same large **scull_fifo_elemsz** as in the previous test. Then start 5 producers in the first terminal (some of them will block, of course). Then start 5 consumers in the second terminal and show that all blocked producers and all consumers then terminate. For example:

```
$ sudo insmod ../driver/scull.ko scull_fifo_size=2 scull_fifo_elemsz=30
$ ./producer p 5
Device (/dev/scull) opened
write: Philippe Meunier
write: Philippe Meunier
write: Philippe Meunier
write: Philippe Meunier
write: Philippe Meunier
Device (/dev/scull) closed
```

With at the same time in another terminal window:

```
$ ./consumer p 5
Device (/dev/scull) opened
read: Philippe Meunier
read: Philippe Meunier
read: Philippe Meunier
read: Philippe Meunier
read: Philippe Meunier
Device (/dev/scull) closed
```

You must provide a single screenshot that shows both Debian terminal windows that show output similar to what is shown above (including the **insmod** command used), followed by the output of the **id** command (in either of the two terminal windows).

5) You should also check that the same thing works fine if you start the 5 consumers before starting the 5 producers (no need to provide any screenshot for this).

6 What to Submit For This Assignment

Remember to put comments everywhere in your code to explain to the CAs what your code is doing! Check that your code still works after adding the comments!

Once both your kernel module and your user-space test programs work, execute **make clean** in both the ~/pmeunier-pa5/driver/ and ~/pmeunier-pa5/src/ directories to delete any extraneous files there (replace my name with your student login name, of course).

In the top level of the same ~/pmeunier-pa5 submission directory, create a PDF file named "screenshots.pdf" that contains:

1) Your full name.
2) The Stevens Honor pledge.
3) The four screenshots you created above: one for each of sections 5.1, 5.2, 5.3, and 5.4. Make sure the screenshots are clearly readable.
4) Also add a short explanation before each screenshot so the Course Assistants know what you are trying to show in those screenshots.

At this point the ~/pmeunier-pa5 directory must contain all the files you were given in the original zip file (with some of them modified by you, of course), plus your PDF file containing the screenshots, nothing more, nothing less.

Create a ZIP file of your submission directory (the **`-r`** option means to zip all the subdirectories recursively: do not forget it; also use your own student login name, obviously):

```
cd; zip -r pmeunier-pa5.zip pmeunier-pa5
```

Once you have correctly created the file "pmeunier-pa5.zip", copy it to the host OS using your shared folder, **double-check its content to make sure it contains everything**, then submit it on Canvas.

Done!

7 Rubric

1) Code for section 3: 60%
2) Code for section 4: 20%
3) Screenshots for section 5: 20%

Important note: you do not get points for screenshots unless the corresponding code is submitted too. Screenshots alone will not get you any points at all. So make sure you double check everything before you submit on Canvas.