Laboratory Assignment #3

Extending *scull*, a *char* pseudo-device

Value: (See the **Grading** section of the Syllabus.)
Due Date and Time: (See the **Course Calendar**.)

Summary:

This is your first exercise that involves writing and debugging code. You will just be modifying the code provided by the textbook author for the ***char pseudo-device scull***, to provide a new type of minor device. The new type of minor device will implement a specific kind of alphabetically-sorted buffer structure, and will make use of the kernel's ***mutex*** locking facilities. This will not require changing all of the existing code, but you should understand it.

Objectives:

- Read and comprehend an existing driver.
- Add a new minor device type to an existing device driver.
- Experience testing and debugging your own changes to kernel module code.
- Learn about mutex kernel locks.

Tasks:

1) Get a copy of the source code, which can be found on the class website.
2) Starting with the provided Makefile, compile the unmodified version of scull and run through the tests described under "Playing with the New Devices" in the LDD3 text.
    a. Change directory to the scull subdirectory.
    b. Use chmod to make the script files scull_load and scull_unload executable, or else you will need to use the sh/bash syntax ". ./scull load" (in csh: "source ./scull_load") to cause the shell to read them in and execute them. (Consider splitting these scripts, as suggested under Advice, below.)
    c. After reading scull_load, execute it (as root) to install the module and create the entries in /dev for the devices.
    d. Create simple scripts and/or programs to test the scull devices (as an ordinary user). A simple, example C program, **`sculltest.c`**, can be

found in the source directory and used as a starting point for more extensive testing of your modifications.

e. After reading scull_unload, execute it (as root) to remove the entries in /dev for the devices, and remove the module.

3) Modify scull to add a new type of device that implements an alphabetically-sorted buffer (SORT) called **scullsort**. The new code should retain all the functionality and minor device numbers of the old scull, with one new added device number for the SORT type. Use the next available unused minor number. Put as much as possible of your new implementation code in a separate file, named **`sort.c`**, and try to make minimal changes to other source files.
4) Base your device on the existing scull pipe device. It should retain the same basic data structures and the same blocking behavior when a process tries to read from an empty device or write to a full device.
5) The read operation should return the number of requested bytes from the SORT buffer in smallest-unsigned-int-first order. That is, consider each byte in the buffer to be a separate element. Therefore, the first byte that is returned should be the byte that when cast to an unsigned integer has the smallest value. Subsequent returned bytes should again have the smallest unsinged integer value. The read operation should return as many characters as requested, up to the maximum that are in the SORT buffer. That is, if there were two writes to the SORT, of “hgfed” and "cba", followed by a read of 5 characters, the value 5 should be returned and the string "abcde" should be copied to the user's buffer. If the next read requests 5 more characters, the value 3 should be returned and the string "fgh" should be copied to the user's buffer.
6) The behavior of an attempt to read from an empty SORT should block or not block, depending on the whether the application opened the SORT with the O_NONBLOCK flag. If O_NONBLOCK is not set, the read operation should block until some data is placed into the buffer. If O_NONBLOCK is set, a read should return immediately with the standard behavior specified for read with O_NONBLOCK when data is not immediately available (see the man page for more details on the expected behavior).
7) The write operation should store the requested data in the scull’s internal data structure(s). If the number of bytes provided by the call is more than will fit into the SORT, and O_NONBLOCK is not set, the call should write as much as will fit, and then block until there more space becomes available (due to subsequent read operations, or other operations that remove data from the file). It should repeat this writing and blocking until the entire string is written, before returning.
8) This leads to a semantic quandary if the operation is interrupted by a signal after it has written a portion of the string. In this case you should return the number of characters written, rather than returning -1 with an EINTR error.

9) There should be just one IOCTL operation, and it should have the effect of emptying the SORT. It should be named **SCULL_IOCRESET**.
10) The open() operation on an existing SORT should not destroy the content.
11) You should retain the other functionality of the pipe, including the ability of a SORT to be shared between processes. To this end, take care that the new code you add follows the locking conventions correctly.
12) As any other design questions come up, for which the precise behavior is not specified here, you may either resolve them according to your best engineering judgment, or discuss them with me.

Test and debug your modified code. You should devise your own tests, to check that the specifications above are satisfied. You may be able to do some of the testing using shell scripts, for example using cp, dd, and I/O redirection, as suggested in the text. However, for thorough it will probably be necessary to write one or more C test programs, using the calls to read, write, ioctl, etc.

Beware that the above semantics differ from those of the scull pipe in a number of subtle respects, including but not limited to the blocking behavior, the IOCTL calls, and the behavior of the open operation.

Advice:

- Get started right away.
- If you encounter problems, ask the instructors, either in the lab, after class, or by e-mail.
- For testing, you can start using scripts and shell commands, like cat, dd. For thorough testing, you may want to write some programs in C, using the system API.
- You will need to perform the following steps.
    a) Get, compile, and test the basic scull module.
    b) Make a copy of pipe.c using the name sort.c, and revise at least all exported names to make them different from those in sort.c.
    c) Add your new file to the object files in the Makefile.
    d) Add any necessary support for your new device subtype to sort.h and main.c.
    e) Revise the read and write routines to implement the semantics according to the requirements above.
    f) Create your own ioctl routine, with just the one command supported.
    g) Remove all left-over (useless) code from the original pipe.c code.
    h) Modify scull_load and scull_unload to create the appropriate device node(s) for your new device(s), or use these as templates to write your own sort_load and sort_unload scripts, that only creates the node(s) needed for your SORT device.

i) Compile and test your code using appropriate test programs/scripts that you have written.
j) Save your tests for the demonstration.
k) You will also want to modify the Makefile to add targets for all of your tests.

- You would be wise to do the above incrementally, maybe starting with just a duplicate of the pipe code (no changes) that will compile and which you can test.

References:

- The textbooks (remember, LDD3 is out of date!)
- The example source files that came with the LDD3 text
- Anatomy of Linux synchronization methods
- The mutex API

Delivery Method:

1) Sign up for a time slot to demonstrate your code to the TA.
2) Tar and gzip your assignment 2 code repository, and send it to both the TA and instructor's emails. Points may be taken off if unnecessary files (such as object files) are sent.
3) Come to your demonstration with printed copies of the files that you modified, with the parts you modified pointed out in some clear way (e.g. marked colored highlighter), including copies of the test scripts and/or programs you used. The instructor will mark up these print-outs during the demonstration, so please make certain you have them.

Assessment:

If you do everything that is required and explain it adequately at the demonstration you will receive a perfect score of 100. Deductions will be made according to the table below. The right-hand column shows the maximum number points that may be deducted for each missed requirements. Bonus items, at the end of the table, may earn points to compensate for some points deducted for missed requirements.
Pay attention to the quality of the tests that you prepare for the demonstration, as well as readability of your code. During the demonstration, I will be looking to verify that you can demonstrate tests for all the required functionality and will also look over your code.

Requirements	Max Deduction
Do you have a paper print-out of your code and tests? Have you submitted your repository to the TA and instructor via email?	-10 and reschedule
Is the paper print-out marked to indicate what parts you changed/added?	-10
Are you are able to compile and load the kernel module?	-100
Can you explain your code, and answer questions about both what you did and why?	-100
Do you have the test scripts or programs ready to go? (no editing during tests)	-10
Does the read operation sort the previously written data correctly? For example, if you write " hgfed" and " cba" but then read back 5 characters, do you get "abcde" back? (not "defgh")	-10
When a read asks for more characters back than are in the SORT, does it return all those that are left?	-10
When a read encounters no data in the buffer, does it block if-and-only if the file was not opened with the *O_NONBLOCK* option?	-10
When a write tries to write more characters than the SORT can currently hold, does the write operation block (looping internally, as necessary) until the entire string has been written, if *O_NONBLOCK* is not set?	-10
Under the similar circumstances, does the write operation return immediately (without writing anything) if *O_NONBLOCK* is set?	-10
Does reading from the SORT consume data?	-20
When the write operation blocks, and then a subsequent read create more space, does the write operation unblock?	-10
Does the **SCULL_IOCRESET** *ioctl()* command empty the SORT? (an alternative ioctl command name could be defined e.g., SCULL_IOCEMPTY)	-20

Does the SORT allow concurrent access? Have you tested it with a concurrent reader and writer? Multiple readers? Multiple writers?	-20
If one process puts data into the SORT, and then closes it, is the data still there when another process opens the SORT to read or update it?	-20
Have you maintained the correct use of locking to protect critical sections? Does the protection cover all operations that access the device data? (reads? writes? IOCTLs?)	-20
Have you preserved the old types of devices when you added your new device, without breaking any of them or introducing possibilities for unchecked incorrect usage?	-10
Do the operations on your new device have any useless code, perhaps left over from cutting and pasting? Code to wrap around the end of the buffer? IOCTLs that don't make any sense for a SORT?	-10
Do you check the validity of all calls on your device? In particular, do you check for validity of IOCTL calls, that any IOCTL's that do not make sense for your device are rejected? (You cannot just modify the existing *scull_ioctl()*. You need to override it with a function specific to SORT.)	-10
Optional Features	**Max Bonus**
Have you written programs for testing?	+5
Do you have any special creative features that you invented, not required by the assignment?	+10
Did you use create a reference manual for your project (e.g., Doxygen)?	+5
Have you used source control (e.g., git)? Do you have an active log? Can you show the changes (i.e., diff) made to the initial source code?	+5
Did you correct/improve the existing scull source code?	+5

Note: the testable functional requirements above are predicated on your code should be free of observable fundamental kernel programming errors, including the following:

- Race conditions
- Unprotected critical sections
- Memory leakage
- Dangling references to freed memory
- Lock usages that permit deadlock
- Unchecked calls to functions that return status, such as kmalloc()
- Practices that risk kernel stack overflow, such as allocation of unpredictable-sized local arrays in kernel functions
- etc.

A minimum of 5 points will be deducted for each of the above errors. More may be deducted depending on the severity of the error.

Changelog:

03Jun14 – It is acceptable to define a new ioctl command name, rather than use **SCULL_IOCRESET**, for this assignment. Based on the observation from Harish, the specified SCULL_IOCRESET ioctl command described in this assignment can be considered to have a different meaning than that of the original scull driver. Given that the commands have differing expected effects, a valid response would be to use a different ioctl command name (e.g., SCULL_IOCEMPTY). If a different ioctl command name is used in your assignment, be sure it is clearly documented.