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 us of the newer mutex lock structure instead of the (older) semaphore lock. This will not require changing all of the existing code, but you should read it all through and 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 kernel module code.
- Learn about mutex kernel locks.
- Learn to use the source versioning-control system 'git'.

Tasks:

1. Get a copy of the textbook source code on the code page of the website. I have also directly linked to the minimum set of files you will need to complete the assignment, but the textbook source may contain extra helpful code.
2. Setup and use git throughout your development. Please create a separate directory for your project – do NOT add your entire kernel source tree to the repository. (If you end up committing your entire kernel source tree at this point, I may take off points.) Also, make sure you are only committing code to your repository – not object files or compiled modules. Please refer to the git class presentation and references for more information on setting up git. I will expect to see commit activity in your git log as you work on the assignment. If you do not wish to use git and would rather use another revision control system like svn, please email me and let me know.
3. Starting with the provided **Makefile**, compile the unmodified version of *scull* as it comes from the text, and run through the tests described under "Playing with the New Devices" in the LDD3 text.
    - You will need to be running the class version of the kernel (3.2.x) for the module to compile correctly.
    - Change directory to the scull subdirectory.
    - 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.)
    - After reading **scull_load**, execute it (as root) to install the module and create the entries in /dev for the devices.
    - Use shell commands and/or short C programs you have written (*e.g.*, see **sculltest.c**) to test the devices (as an ordinary user).

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

4. 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 make the minimum changes necessary to the other source files and the Makefile.
5. Base your device on the existing scull *pipe* device. It should retain the same basic data structure, and the same blocking behavior when a process tries to read from an empty device or write to a full device.
6. The read operation should return the number of requested bytes from the SORT buffer in smallest-unsigned-int-first order. It 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.
7. 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).
8. The write operation should append the string to the end of the SORT. 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*.
9. 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 *errno* equal to *EINTR*.
10. There should be *just one* IOCTL operation, and it should have the effect of emptying the SORT. It should be named SCULL_IOCRESET.
11. The *open()* operation on an existing SORT should *not* destroy the content.
12. 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.
13. 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.
14. Test and debug your modified code. You should devise your own tests, to check that the specification 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 direct 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 behavor, the IOCTL calls, and the effect of the open operation.

Advice:

- Get started right away.

- If you encounter problems, ask the instructors, either in the lab after class, by e-mail, or by telephone.
- The file sorttest.tgz (found on the code class page) contains a test program for the SORT device, which you should test against your implementation. In order to use this, you will need to pay attention to the following details:
    a. The `#ifdef __Kernel__ ... #endif` needs to be added to a section of *scull.h* in order to be able to reference the *ioctl()* command names from the application program *sorttest.c* without pulling in a bunch of unwanted stuff from the kernel header files.
- For testing, you can start using scripts and shell commands, like `cat`, `dd`, but this is limited in the scope of what can be tested. For thorough testing, you will need to write some programs in C, using the system API. For a starting point, see **sorttest.c**, in the tarfile mentioned above.
- You will need to perform the following steps.
    a. Get, compile, and test the basic scull module.
    b. Create a new git repository.
    c. Make a copy of `pipe.c` under the name `sort.c`, and revise at least all exported names to make them different from those in `sort.c`.
    d. Add your new file to the object files in the `Makefile`.
    e. Add any necessary support for your new device subtype to `sort.h` and `main.c`.
    f. Revise the read and write routines to implement the semantics according to the detailed requirements above.
    g. Create your own `ioctl` routine, with just the one command supported.
    h. Remove all left-over (useless) code.
    i. 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 create the node(s) needed for your SORT device.
    j. Compile and test your framework/prototype, using appropriate test programs/scripts that you have written.
    k. When you have a working solution, save your complete configuration, and then modify all the files to use kernel *mutexes* instead of kernel *semaphores*.
    l. Save your tests, for the demonstration.
    m. 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 instructor, using the Doodle poll on the class website.
2. Tar and zip up your assignment 2 code repository, and send it to both the TA and instructor's emails. Please leave your git files in the repository. Points may be taken off if unnecessary files (such as object files) are left in your repository.

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, but will also read 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/wrote?	-10
Have you used git? Do you have an active git log?	-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 you did it that way?	-100
Do you have the test scripts or programs ready to go? (no editing during tests)	-10
Does the read operation behave in an alphabetically-sorting manner, at the character level? 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 the data?	-20
When the write operation blocks, and then a subsequent read creates more space, does the write operation unblock?	-10
Does the SCULL_IOCRESET *ioctl()* command empty the SORT?	-20
Does the SORT allow concurrent access by multiple processes? 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	-20

protection cover all operations that access the device data? (reads? writes? IOCTLs?)	
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
Have you replaced the semaphore by a mutex?	-10
Optional Features	Max Bonus
Have you written C programs (not just scripts) for testing?	+10
Do you have any special creative features that you invented, not required by the assignment?	+20
Did you use Doxygen to create a reference manual of your project?	+5

Note: the testable functional requirements above are predicated on you 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.

I have taken five (5) points off for each of the above errors, any one of which is serious enough to cause a kernel crash.