Programming Assignment #4

Writing a simple parallel port device driver

Value: *(See the **Grading** section of the Syllabus.)*
Due Date and Time: *(See the **Course Calendar**.)*
Summary:
This is your second exercise that involves writing and debugging code. The main new feature is that you will need to use the kernel timer and/or work queue mechanism to schedule periodic output. You will be writing a new device driver that uses the parallel port to drive a 7-bar LED display device. You may find useful examples and bits of reusable code in the examples provided by the textbook author, such as *jit*, *jiq*, *scull*, *scullp*, and *short*. However, unlike the preceding assignment, where you were adding onto an existing driver and making very few changes to it, this time you need to write a new module *from scratch*.

Objectives:

- Read and comprehend an example parallel port driver "short".
- Experience interfacing directly to real hardware.
- Improve your skill at testing and debugging your own kernel module code.
- Learn how to implement periodic polling behavior in a device driver, using a kernel timer and/or work queue.
- Apply mutual exclusion and synchronization methods.
- Using git.

Tasks:

1. Find the source code for the short driver on the course code page.
2. Make sure the “parport” module is not loaded or compiled into your kernel. It is not enough to simply unload the parport module, as it leaves the ports in an inconsistent state. You must reboot the machine and keep the parport and all related modules/code from loading in the first place. One way to do this is to edit the /etc/modprobe.d/blacklist.conf file and add the following lines:

   ```
   blacklist ppdev
   blacklist lp
   blacklist parport_pc
   blacklist parport
   ```

   Then reboot the machine. Alternatively, you can recompile the kernel to remove all the parport code and modules.

3. Attach one of the LED test devices provided in the lab (similar to the one described in the LDD3 text under "An I/O Port Example" — see description below) and attach it to the 25-pin parallel port D-connector of the machine on which you are working. The entire figure 8 should light up. If you only see the middle “-“, it means the parport code has still claimed the parallel port—see step #2 above.
4. Use the Makefile provided with the original version of *short*, and test it, as described on pages 246-248 under "A Sample Driver" in the text and demonstrated in class.
5. Check that all the LEDs on the device are working.
6. Using any relevant bits of code you like from *short*, write a new device driver of your own, called *ledclock*, that displays a countdown timer on the LED test device. The idea is that you initialize the device to a time value and it then starts counting. When it reaches a given limit, called the

modulus, it either stops or wraps around and reinitializes to zero. Whether it stops or wraps around is governed by a user-controllable parameter.

7. For example, if the modulus is set to 5, it should count either `0, 1, 2, 3, 4` and stop, or `0, 1, 2, 3, 4, 0, 1, 2, ...`.
8. Since the hardware can display only one digit at a time, the driver must display multi-digit numbers by sequencing through the digits from beginning to end, pause, and then repeat.
9. The time value should be displayed in seconds. For example, a time value of 12 seconds should be displayed as "12 " and then (ideally) one second later should change to "13 ". The spacing is intended to indicate that there should be a short pause on each digit and a longer pause before repeating the time. At lower speeds (see variable speed below) it will take longer than one second to complete the digits, so you may skip some output values.
10. Besides displaying the time, your driver should allow reading and writing the time value as an unsigned integer using the *read()* and *write()* methods. The *write()* operation should have the effect of setting the modulus to that value, and the current time to zero (upon which the clock resumes advancing), and the *read()* operation should return the value that the clock has at the time the *read()* method is called.
11. The representation of the time value for *read()* and *write()* should be interpreted as an unsigned integer. For example, to read a time value from the clock you would do the following:
12.
```
int fd;
unsigned int T;
fd = open ("/dev/ledclock", O_RDWR, 0);
read (fd, &T, sizeof (unsigned int));
```
13. Writing a new time value of *300* would be done as follows:
14.
```
T = 300;
write (fd, &T, sizeof (unsigned int));
```
15. *Note that return value checks for system calls are only left out of the examples above for expository purposes.* A real application would need to check the return values, and the driver needs to handle erroneous usage by returning the proper value and setting *errno* appropriately.
16. If a user makes a *read()* or *write()* call with lengths other than *sizeof (unsigned int)*, the user call should return *EINVAL* without performing the read or write operation.
17. There will be some parameters, including whether the clock wraps around or stops when it gets to the modulus value, how long the LED stays on (dwells) for each digit, and the durations of the pause between successive time values. You should provide a compile-time configurable default symbol for all such parameters, plus a driver parameter for start-up time configuration, and finally *ioctl* commands to allow each of these parameters to be changed at run time.
18. The device should start up blank, and stay blank until a value is written to the timer. You should also provide an *ioctl* call to start the device displaying, and one to stop it. Similarly, you should provide an *ioctl* to start and stop the timer from counting, so that the device will continue to display whatever it was last displaying, without changing.
19. There may be wiring differences between the various LED devices, which cause differences in the correspondence between bit positions in the output byte and LED element positions in the display. Therefore, instead of hard-coding the correspondence of digits to display bytes, it would be better to use a table and provide a way to replace the default table by a custom one via an *ioctl* call.
20. Of course, you will need to implement *init_module*, *cleanup_module* and the other required methods, including *open* and *release*.
21. Your *cleanup_module* method is required to stop all display timers/work-queue handlers from rescheduling themselves. This could be done by cancelling them, or you can write a flag that tells them not to reschedule themselves, and then block in the *release* call until each timer/work-queue handler executes one last time.
22. If you start from *short* be sure to put your own name and comments in, and delete any unused code, before you turn in your assignment.
23. Test and debug your modified code. You should devise your own tests.

Advice:

- Apply an incremental development approach. That is, develop your module in stages,test each stage as you go, and adding more functionality at each stage. For example, you could start out with just enough to statically display one digit (*e.g.* a few functions such as the *init_module*, *cleanup_module*, *open*, *release*, and *write* methods from *short.c*). You could then add a periodically rescheduled timer or work-queue item to update the digit displayed. You could then add in the code to read and write the timer value, and to convert between binary unsigned integer format and a displayable sequence of digits. You could then add in some *ioctl* functions, etc. At each stage you add in a bit of functionality, test it, and debug it. That way you always have solid foundation, and verify that you understand what you are doing before you have written a lot of code. Also, if you run out of time, you always have something that is at least partially functional to turn in when the assignment comes due.
- You are free to work out the unspecified details, according to your own best judgment. However, please note: The **quality** of such decisions will be taken into account in the assessment of your work. Keep in mind the objectives of the assignment. You are trying to make a "useful" device, as well as to develop your knowledge and skills in writing Linux device drivers. When you make design choices, choose in the direction of providing greater functionality and convenience for users, and in the direction of using more of the techniques we have been studying. If you have doubts, discuss design decisions with me. If issues that seem to be of general interest to the class come up in such discussions, they will be e-mailed or posted them on the course web pages so that other students may also benefit.
- This requires an event-driven design, based on a kernel timer or work queue. When doing event-driven program, the best practice is to design in terms of a state machine. For example, you can assume the driver timer state has having at least the following components:
  - Uninitialized or initialized
  - If initialized: the modulus
  - If initialized: the current value of the timer
  - If initialized: ticking or not
  - If initialized: displaying or not
  - If displaying: (a) displaying a digit; (b) pausing between digits
- Various events, such as the *write* operation and *ioctl* operations, and the timer handler execution, can cause state changes. Work out what are the valid state changes for each event.
- Remember to use appropriate locking mechanism(s) to protect the state transitions. Remember that whatever you use cannot require the timer-handler code to block.
- Take care with the code for shutting down timers, including both the caes of explicit shutdowns via IOCTL and implicit shutdowns via *release()* method or module removal. If this is not done right, there is a possibility of a timer re-arming itself after you think you have stopped it. Be especially careful if you decide to use more than one timer, or a work queue item and a timer.
- If you encounter problems, please ask for help, either in the lab after class, by e-mail, or by telephone. I will try to help. If we learn something that may be useful to the entire class I will send the entire class an e-mail with the information.

Pitfalls to Avoid

- Beware of compiling the standard parallel port driver into your Linux kernel. If you have the standard driver installed, when your driver tries to register it will fail, because the standard driver already has claimed the right to serve the parallel port.
- Beware of false assumptions and misunderstandings based on the prior assignment or the example code from the LDD3 book. For example, the "short" example includes an interrupt handler, but you do not need any interrupt handling. The "short" example also includes a lot of other functionality that you do not need, including input. What you do need to learn from that example is how to claim the right to serve as driver for the device, and how to output to it. Similarly, the "jiq" example is helpful for seeing how to use a timer, but it includes a lot of other stuff that will not be useful for this assignment.

- Beware of thinking about the driver as "looping" to accomplish the clock-ticking and displaying functions, or "sleeping" and waking up. You do not have a thread that you can dedicate to doing this, so you need to implement these functions in an event-driven fashion, based on a kernel timer. In the event-driven style iteration is implemented by having the timer handler reset the timer, which will cause it to execute again later.

Helpful Resources:

- short example, for code to put a value out to the parallel port.
- jiq.c for code to use kernel timers and work queue items

Delivery Method:

1. Sign up for a conference time slot to demonstrate and explain your code to the instructor, using the Doodle poll listed on the class calendar page.
2. At the conference, turn in the following, as hard copy:
   - o Copies of the files that you modified, with the parts you modified pointed out in some clear way (*e.g.* marked colored highlighter).
   - o Copies of test scripts and/or programs you used.
3. Also e-mail the instructor and TA the electronic copies of the above. You will lose points if this is not submitted before the demo.

The LED Device:
For this exercise you will need a hardware device to connect to the parallel port of your machine, similar to the device described in the text, in the section under the heading "A Sample Driver". There are not be enough of these devices for all the machines in the lab, and certainly not enough for you to take one from the lab. As an alternative, you can construct a device of your own using parts that you can buy from from Radio Shack.

If you are working at home and want to check out one device for your personal use, please see me to sign an accountability form. The other devices should be left in the lab.
If you are interested in making your own unit, here are the parts we used:

- a "printer cable" having "male" 25-pin D connectors on each end
- a 25-pin female D connector
- an 8-element LED digital display module (has seven bar elements, plus a decimal point)
- one 150-ohm resistor
- a socket to hold the LED module
- a small piece of perf-board to hold the components
- glue, solder, a soldering iron, small bits of wire

You use the perfboard, solder, and wire to hook up one of pins 2-8 of the 25-pin connector to each of the seven pins of the LED bars. You then hook up pin 20 of the 25-pin connector to the common (ground) connector of the LED display, with the 150 Ohm resistor in between. The 25-pin connector is plugged into one end of the parallel cable, and the other end goes into the computer. The hardest part is finding a 25-pin connector and attaching it to the perfboard. If you are doing this yourself you can attach a cable-end 25-pin connector to the perfboard with wires. If you do this, consider using thread to sew down the wires to the board for strain relief, or the solder connections will break. You can look at the various examples in the lab for details.

Assessment:
Your work will be judged primarily on well it performs during the demonstration, and how well you can explain what you did. Remember that the development of tests is part of the assignment. I will also ask

you to show me through the code, and explain it. The ideal is for your code to be as simple as possible, for the functionality you implement. Including unused or inappropriate bits of code from the examples in the text will be interpreted as an indication that you either were not willing to make the effort to delete useless code, or that you do not understand the code well enough to decide what is needed and what is not. The latter pitfall is sometimes called **cargo-cult programming**.
When you demonstrate your code you should expect to show at least the following:

- When it starts up the display should be blank.
- When you write a time to the device, it should begin displaying the time, in a circular fashion, with a blank pause at the end before it cycles around and counts up again.
- It should wrap around to zero after displaying the value before the modulus, and then either stop or continue, according to how you have set it via the *ioctl()* operation.
- You should be able to read back the time, as an *unsigned int*, from the device.
- You should be able to change the speed of the display, so that it sequences through the digits faster and slower.
- You should be able to pause and restart the timer
- You should be able blank the display (and stop any kernel timers and/or work-queue item) without removing the driver, via and appropriate *ioctl()* call.
- After completion of *rmmod* the device should become blank.

You will also be asked to show where, in your code, you provide for the following:

- Appropriate control over timing. (In the past, some students have failed to use an appropriate kernel scheduling mechanism, and have instead tried to code this using a loop. That is *not acceptable*.)
- Protection of critical sections, of two kinds:
    a. Between multiple user processes accessing the device (e.g, for *open*, *read*, *write*, *ioctl*)
    b. Between a timer and or work-queue handler and a user process
    c. Between multiple timers/work-queues if you have more than one. (However, it is strongly recommended that design your solution to get by using just one kernel timer.)
- (Take care here. In the past, many students have forgotten to protect some critical sections, or used the wrong kind of lock, e.g., a semaphore where a spinlock is needed, or vice versa. Others have used the right kind of lock, but created flows of control that could lead to deadlock.)
- Safe shutting down of any kernel timers or work-queues you are using, without races. (In the past, students have often gotten into a race between the shutting down of a kernel timer and the timer rescheduling itself. This is more likely to be a problem if you have more than one, so try to design to use only one.)

More items may be added to the above check-list, in response to e-mails and in-class discussions.