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 programming exercise involves writing a kernel module to drive an external parallel port device. The device is a 7-segment led display wired to allow the parallel port hardware and software to light the individual segments. The kernel module described in this assignment will use the time/timer facilities of the kernel to schedule time-based control of the LED device. Useful code examples are provided by the textbook author, including; *jit*, *jiq*, *scull*, *scullp*, and *short*. While you may certainly use code from the examples, you should be aware that unlike previous character assignment you will not be provided an initial template as part of the assignment. The code and design of your driver will be up to you.

Objectives:

- Read and comprehend an example parallel port driver (i.e., `short`).
- Experience communicating with external hardware devices.
- Improve processes for testing and debugging kernel code.
- Learn techniques to achieve time-related hardware I/O using the kernel's time/timer subsystems.
- Design and implement mutual exclusion and synchronization techniques.

Tasks:

1. Find the source code for the **`short`** driver on the course's webpage.
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 modules/code that touch the parallel port hardware from loading in the first place.
    - One way to do this is to add the following lines to **`/etc/modprobe.d/blacklist.conf`**:

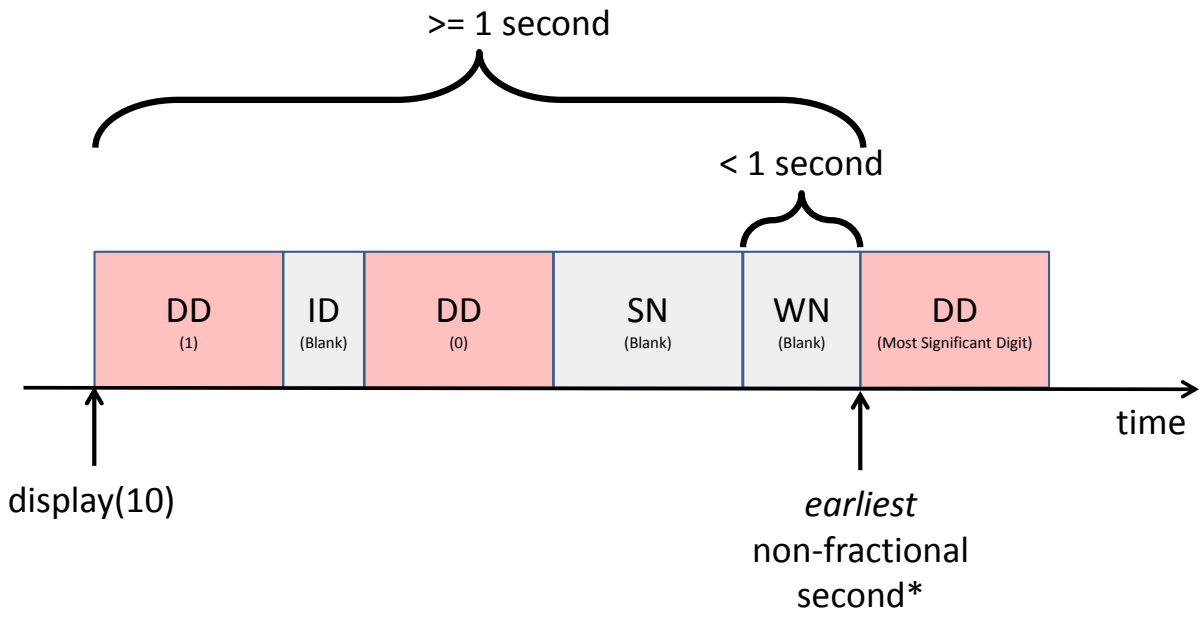
      ```
      blacklist ppdev
      blacklist lp
      blacklist parport_pc
      blacklist parport
      ```

      and reboot the machine.

    - Alternatively, you can recompile the kernel to remove all the **`parport`** code and modules.

    - Yet another alternative is to add **`modprobe.blacklist=modname1,modname2,modname3`** to the kernel command line.

3. Attach one of the LED devices (similar to the one described in the LDD3 text under "An I/O Port Example" — see description below) to your machine's 25-pin parallel port connector (DB-25). If you only see the middle '–', it typically indicates a problem that the **`parport`** code has claimed the parallel port hardware (see #2 above).
4. Test the device with the *short* module (described on pages 246-248 under "A Sample Driver" in the LDD3 text).
5. Verify that all LED segments on the device are working. The devices have been used quite a bit over the years and may have become faulty.
6. Using any relevant bits of code you like from *short*, write a new device driver of your own, called *ledclock*, that displays a second counter 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 (whether it stops or wraps around is governed by a user-controllable parameter described later).
    - 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, ...`
7. The displayed time value represents a non-fractional second value from the time instant the device's `write()` file operation is called. Since the hardware can display only one digit at a time, the driver must display multi-digit numbers by sequencing through the individual digits by turning on/off the LEDs for different lengths of time as follows:
        1. **display_digit_time (DD)** - the length of time to light the LEDs to display a digit (often referred to as dwell time)
        2. **inter_digit_time (ID)** – the length of time the LEDS are off between digits of a given number (e.g., if the number is 10, this is the time the LEDs are off between 1 and 0)
        3. **succ_number_time (SN)** – the length of time the LEDs are off between numbers
    - The time instant of the given second (number) being displayed should be the time instant the LEDs transition from off to on for displaying the most significant digit of the current number.
    - Remember that the time being displayed indicates wall-clock time seconds from the time instant `write()` is called on the device; therefore, depending on the on/off times listed above, skipping the display of some second value may be necessary.



*Note that the next value (number) to display will depend on the various parameters (e.g., modulus)

8. The driver should allow reading and writing time values 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.
9. 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:

```
int fd;
unsigned int T;
fd = open ("/dev/ledclock", O_RDWR, 0);
read (fd, &T, sizeof (unsigned int));
```

10. Writing a new time value of *300* would be done as follows:

```
T = 300;
write (fd, &T, sizeof (unsigned int));
```

Note that return value checks for system calls are only left out of the examples above for expository purposes. A real application would most likely check the return values, and the driver needs to handle erroneous usage by returning the proper value and setting *errno* appropriately.

11. 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.
12. Various parameters are used in this assignment, including whether the clock wraps around or stops when it reaches the maximum value, how long the LED stays on (dwell time) for each digit, and the blank time duration between displaying successive values/digits. 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 *(*or *sysfs files)* to allow each of these parameters to be changed at run time.
13. The device should start up blank, and stay blank until a value is written to the timer.
14. **(PAUSE ioctl)** An ioctl command (or sysfs file) should be implemented to pause counting. If the display is currently blank (e.g., due to inter-digit delays), the previously displayed digit should be displayed while the device is in the paused state (unless the display is turned off (see below)).
15. **(WRAP ioctl)** An ioctl command (or sysfs file) should be implemented to allow the user to specify the wrapping condition. That is, assuming a modulus of *n* whether counting should wrap around to zero after displaying (*n* – 1) or whether the counting should stop at (*n* – 1).
16. **(DISPLAY ioctl)** An ioctl command (or sysfs file) should be implemented to turn off/on lighting of all the LEDs. The only noticeable effect of this ioctl is that the LEDs should not light. Other functionality should continue with the same effects as if the ioctl to turn off the LEDs were never called. Put another way, when the LEDs are turned off using this ioctl, the driver should not lose track of the display state with respect to wall-clock time. For instance, after the LEDs are turned back on (from an “LEDs off” state) the device should display exactly the same as if the device were never turned off.
17. 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 (not required, but make sure for the demo you know which LED works for your driver).
18. Of course, you will need to implement *init_module*, *cleanup_module* and the other required methods, including *open* and *release*.
19. 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.

20. If you start from *short* be sure to document the code you used from short and your contributions. Any unused code from short should be removed, before you turn in your assignment.
21. Test and debug your code. You should devise your own tests that sufficiently demonstrate correct functionality of all requirements listed in this write-up. The tests should be such that the demo can be performed in a reasonable amount of time. In particular, the amount of interaction between you and your computer should be small.

Advice:

- Apply an **incremental** development approach. That is, develop your module in stages, test each stage as you go, and add 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 on the course web pages so that other students may 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 (c) pausing between numbers
    - …
- 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 cases of explicit shutdowns via IOCTL and implicit shutdowns via *release()* method or module removal. If this is not done correctly, there is the potential that a timer may re-arm itself after the logic of your code assumes that it is stopped. Be especially careful if you decide to use more than one timer, or a work queue item and a timer.

Common Pitfalls to Avoid

- Beware of compiling the built-in parallel port driver into your Linux kernel. If you have the built-in driver installed and your driver tries to take ownership of the parallel port hardware it will fail since the in-kernel driver already has claimed control of the parallel port hardware.
- Beware of the different requirements of the example code in the LDD3 book and those of this assignment. 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 control to serve as driver of the parallel port hardware, and how to change the pin voltages. 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 should not have a thread that you dedicate to doing this, so you need to implement these functions in an event-driven fashion, based on a kernel timer.

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 time slot to demonstrate and explain your code to the TA in class on the due date.
2. At the demo time, turn in the following, as a hard copy:
    - Copies of the files that you modified, with the parts you modified pointed out in some clear way (*e.g.* marked colored highlighter).
    - Copies of test scripts and/or programs you used.
3. Also e-mail the instructor and TA the electronic copies of the above **before** the demo.

LED Device Hardware:

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 may 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 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:

- "printer cable" having "male" 25-pin D connectors on each end
- 25-pin female D connector
- 8-element LED digital display module (has seven bar elements, plus a decimal point)
- one 150-ohm resistor (may vary depending on your LED module)
- 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. Then, hook up pin 20 of the 25-pin connector to the common (ground)

connector of the LED display, with an appropriate 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 may quickly fail. You can examine the already built devices in the lab as a reference.

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**. You will also be asked to give a tour of your code, adequately explaining it. Including unused or inappropriate bits of code from the examples in the text will be interpreted as an indication that you do not understand the code well enough to decide what is needed and what is not. A pitfall often referred to as **cargo-cult programming**.
When you demonstrate your code you should expect to demonstrate **at least** the following (additional functionality listed in the write-up should work as specified and may be asked to be performed during the demo):

- 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.
- Proper **wrapping** behavior in accordance with the driver parameters.
- Ability to **read** back the time, as an *unsigned int*, from the device.
- Ability to change the various **timing parameters**.
- You should be able to **pause** and restart the timer.
- You should be able **blank** the display without removing the driver, via an appropriate *ioctl()* call.
- After your module is **removed** the no LEDs of the device should be lit.

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

- 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 busy-waiting (this 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 to design your solution to use only 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 mutex 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 timer.

Acknowledgement:
A large number of improvements to this write-up resulted from numerous conversations with Harish Chetty.

Changelog:

18Jun14: Clarifications to the on/off and pause ioctl commands. Other minor, miscellaneous changes wer made to the wording/formatting to improve the clarity of the driver's expected behavior.

19Jun14: Remove requirement for countdown functionality. The device is only expected to count up and wrap.

23Jun14: Clarify the meaning of the time instant of a non-fractional second and its relationship to the time instant the device's `write()` file operation is called.