CMPSC 311 Assignment 3: Userspace Device Driver

Out: February 25, 2014 Due: March 17, 2014

Purpose

This assignment is designed to introduce you to writing low-level driver code for a realistic device. Given the hardware specification for a video device, you will create a device driver which communicates with virtual hardware on a bit-and-byte level, providing a simple abstraction for higher layers to use.

Description

In this assignment you will write a device driver for an array of low-resolution displays, which we will call a MultiTron.[1] The MultiTron consists of a rectangular array of displays which will be treated as one large virtual screen. Each display has a resolution of 256×128 pixels and a color depth of 8 bits (i.e., each pixel is an 8-bit value). Display IDs are assigned by the MultiTron hardware from left to right and top to bottom. For example, this array would appear as a 1280×384 screen:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

The device driver implementation will provide an API of several functions which will be called by the layer above it. When a function is called, your driver will send the appropriate commands to the physical hardware to perform the requested operation. It is up to you to determine how best to implement these functions within the constraints laid out in the assignment.

Your device driver will provide a virtual screen which uses the entire array of small displays. When you connect to the MultiTron, it will communicate to you that r rows and c columns of displays are attached. This means that your virtual screen will appear to be $(256 \times c)$ pixels wide and $(128 \times r)$ pixels high. Your driver functions will accept coordinates `x` and `y` which represent a point within this large virtual screen.

All communication with the MultiTron hardware is done through a single operation interface. This interface is defined by the following function call:

```
int tronctl(uint32_t op, void *data);
```

This function accepts a MultiTron operation `op` and a pointer to a buffer that may be read/written depending on the requested operation. The return value is 0 if the operation executes successfully, and nonzero if there was an error. The structure of the `op` parameter is given in Table 1.

[1] Apparently the word JumboTron® is trademarked by Sony. Who knew?

Bits	Width	Field	Description
0–6	7	Display ID	Selects a display to perform the operation on
7–11	5	Opcode	Opcode of the command to be executed
12–24	13	(Reserved)	Unused bits; must be set to 0
25–31	7	Scanline	Y coordinate of the operation

Figure 1: MultiTron instruction format

Opcodes

The following opcodes are recognized by this hardware:

- `MTRON_POWERON`: Powers on the MultiTron hardware and retrieves information about the size of the array. The display ID and scanline instruction fields should be set to 0. The `data` parameter should point to a buffer which will receive two 8-bit values: first the number of rows in the array (3 in the example drawn above), and second the number of columns (5 in the example). Each display will be automatically initialized to blank pixels (`0x20`) by the hardware.

- `MTRON_POWEROFF`: Powers off the MultiTron hardware. The display ID and scanline instruction fields should be set to 0, and the `data` parameter is ignored.

- `MTRON_READ_LINE`: Retrieves the pixel data for one scanline (row) on the given display, as indicated by the display ID and scanline fields in the instruction. The `data` parameter should point to a buffer which will receive 256 pixel values, from left to right.

- `MTRON_WRITE_LINE`: Sets the pixel data for one scanline (row) on the given display, as indicated by the display ID and scanline fields in the instruction. The `data` parameter should point to a buffer containing 256 pixel values to be written to the display, from left to right.

Simulator

The simulator program provided will set up a virtual MultiTron, read a *workload file* of commands, and call your driver accordingly. This program is used as follows:

```
Usage: ./simulate [OPTION]...
Runs the workload from standard input on a virtual MultiTron.
  -h         show this help
  -v         output verbose log data to stdout
  -o FILE    (additionally) output log data to FILE

To read a workload from a file, put it on standard input:
  ./simulate -v < workload.dat
  OR
  cat workload.dat | ./simulate -v
```

Workload files are provided with the source code to test driver functionality, along with a script called `verify` which will run each of the workloads and ensure that the output matches a reference implementation. Your driver will be checked for correctness against these workloads when it is being graded.

The simulator will output a lot of information if you enable the “verbose” option with `-v`. You can use this to debug your driver. To view this output more easily, pipe it through the `less` program:

```
./simulate -v < workload.dat | less -S
```

When using `less`, you can press the `h` key for help and `q` to quit.

Alternatively, you can output the log data to a file, which can then be opened in an editor or a pager such as `less`:

```
./simulate -v < workload.dat > logfile.txt
```

Procedure

1. Log into your virtual machine and open a terminal emulator.
2. Download the starter code and extract the tarball as you did in the previous assignment:

   ```
   http://www.cse.psu.edu/~djp284/cmpsc311-s14/docs/assign3.tar.gz
   ```

   This will create a directory `assign3` containing these files:

   - `driver.h`: Header file containing prototypes for the functions you will be implementing and a struct to store your driver's state. Fill out the struct as needed, but *do not* change the function declarations; they define the API you will be providing!
   - `driver.c`: Driver implementation. This is where you should write your code.
   - `mtron.h`: Header file defining constants and functions needed by the hardware.
   - `libmtron.a`: Virtual hardware implementation, provided as a static library. (This is the 64-bit version; on a 32-bit machine, edit the Makefile to include `libmtron32.a` instead.)
   - `simulate.c`: Program which reads a workload file of display commands from standard input and calls your implementation to handle them.
   - `verify`: Shell script to run some basic unit tests and verify the output of several workloads.
   - `Makefile`: Makefile for the entire project. You can edit this if needed.
3. Comment all of your code as you write it to explain what the code is doing.
4. Implement the following functions as defined in the interface header file `driver.h`, placing the implementations in `driver.c`:

   - `mtron_init(mtron)`: Powers on the MultiTron and initializes the `struct multitron` passed to the function as needed.
   - `mtron_destroy(mtron)`: Powers off the MultiTron.
   - `mtron_getpixel(mtron, x, y, color)`: Retrieves the color of the pixel at virtual coordinate (`x`, `y`) and stores it in the `color` variable.
   - `mtron_putpixel(mtron, x, y, color)`: Sets the color of the pixel at virtual coordinate (`x`, `y`) to `color`.
   - `mtron_getrect(mtron, x, y, w, h, buf)`: Retrieves the pixel data from a rectangular portion of the virtual screen. The top-left corner of the rectangle is (`x`, `y`), and the width and height are `w` and `h` respectively. Pixel data should be written into the `buf` array from left to right and top to bottom.
   - `mtron_putrect(mtron, x, y, w, h, buf)`: Draws the image stored in the `buf` array on the screen at (`x`, `y`), where the width and height of the image are `w` and `h` respectively. Pixel data in the array is stored from left to right and top to bottom.

Any function with an `int` return type should return 0 for success and nonzero if there is an error. If any parameter is out of bounds (a pixel that is off-screen, or a rectangle that is partially off-screen), the function should do nothing and return immediately with an error value.

Note: the `getrect` and `putrect` functions could be written just by calling `getpixel` and `putpixel` repeatedly, but the result would be extremely inefficient (many calls to `tronctl`). To receive full credit for these functions, you will need to make good use of the fact that you can read/write entire lines in one operation.

Submission

1. Run `make clean` to remove any build products (binaries, object files, etc.) from the directory.

2. Create a tarball containing the `assign3` directory, complete with the source code you have written. The tarball should be named `LASTNAME-psuid-assign3.tar.gz`, where `LASTNAME` is your last name in all capital letters and `psuid` is your PSU email ID (the one you would use to log into ANGEL). For example, if the instructor were submitting this assignment, he would run:

   ```
   $ tar -cvzf POHLY-djp284-assign3.tar.gz assign3/
   ```

3. Verify that the tarball contains all of your source code files by checking a listing:

   ```
   $ tar -tvzf LASTNAME-psuid-assign3.tar.gz
   ```

4. Before sending the tarball, test it in a clean temporary directory using the following commands:

   ```
   $ mkdir /tmp/assign3test
   $ cp LASTNAME-psuid-assign3.tar.gz /tmp/assign3test
   $ cd /tmp/assign3test
   $ tar -xvzf LASTNAME-psuid-assign3.tar.gz
   $ cd assign3
   $ make
   $ # ... now run any commands needed to test your program
   ```

5. Send the tarball in an email with the subject line "311 submission" to both the instructor (`djpohly@cse.psu.edu`) and the TA for your section (listed on the course website). Put your own email in the Cc: line so you can be sure the email was sent properly and contains everything. **YOU ARE RESPONSIBLE FOR DOING THIS CORRECTLY.** Lost emails, emails not sent to both the instructor and TA, bad tarballs, and missing files are the student's responsibility and will be treated as late or missing. There will be no exceptions.

   This email should be sent by 11:59pm of the due date of the assignment; late submissions will be accepted up to **TWO** days late, with a penalty of 10% for each day.

Bonus Points

Modify the driver so that the current pixel data from all of the displays is saved to the file `multitron.state` when the driver is destroyed, and the data is loaded from this file (if it exists) when the driver is initialized. In essence, make the display data persistent across disconnects.

Reminder

As with all assignments in this class, you are prohibited from:

- Copying any content from the Internet.
- Discussing or sharing ideas, code, configuration, text, or anything else with others in the class.
- Seeking significant help from anyone inside or outside the class other than the instructor and TAs.

Likewise, you are responsible to protect your own code from falling into the hands of others. Consulting online sources is acceptable, but under no circumstances should *anything* be copied. Failure to abide by this requirement will result in dismissal from the class as described in the course syllabus.

Above all: when in doubt, be honest. If you do for any reason get significant help from a source, online, in person, etc., *document it in your submission* before we have to ask you about it. This will go a long way!