CMPSC 311 Assignment 3: Userspace Device Driver

Out: February 17, 2015 Due: March 6, 2015

Purpose

This assignment is designed to introduce you to writing a layer in a system and providing a higher-level abstraction of a lower-level interface. Given the interface specification for a piece of virtual hardware, you will create a device driver which communicates with the hardware on a bit-and-byte level, providing a simpler interface for higher layers to use.

Description

In this assignment you will write a device driver for a fictitious 3D printer known as a "TipTap." The TipTap has a printing nozzle that can be moved along the X, Y, and Z axes above a print bed, shown in Figure 1. While moving, it can extrude one of a number of different colored materials to print a three-dimensional object. The nozzle is also equipped with a small laser probe which can determine the amount of empty space directly underneath it. This probe allows the TipTap hardware to support a variety of different print sizes, as well as to function as a 3D scanner.

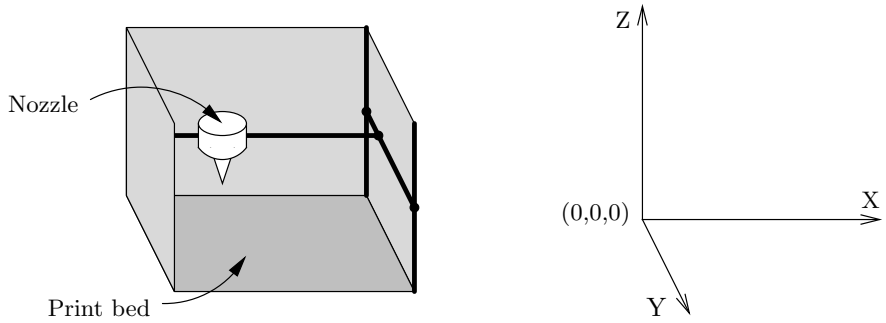

Figure 1: Structure and axes of the TipTap

Device driver API (your code)

Your code will provide an API of several functions which can be used by a printing application for tasks such as initializing the TipTap, positioning the nozzle, printing layers of a three-dimensional model, and powering the device off. When one of these functions is called, your driver will send the appropriate low-level instructions 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.

The driver function specification is as follows:

- `tiptap_init(tt)`: Powers on the TipTap and initializes the `struct tiptap` passed to the function as needed.
- `tiptap_destroy(tt)`: Powers off the TipTap and releases any resources that were allocated in `tiptap_init`.

- `tiptap_moveto(tt, x, y, z)`: Moves the print nozzle to the location (`x`, `y`, `z`). The function should fail without doing anything if `x` or `y` is outside the width or height of the print bed, or if `z` is higher than the initial home position.

- `tiptap_getpos(tt, x, y, z)`: Retrieves the current position of the print nozzle, placing the coordinates in the output parameters `x`, `y`, and `z`.

- `tiptap_printlayer(tt, x, y, w, h, materials)`: Prints one layer of a model at the current Z position, then *moves the nozzle one unit higher in the Z direction* if possible. (This allows an entire model to be printed by repeatedly calling `tiptap_printlayer`.) The layer should be printed starting at the given `x` and `y` coordinates, with the specified width `w` and height `h`. The `materials` parameter points to a $\texttt{w} \times \texttt{h}$ array specifying which material should be printed at each position (or 0 for none), from left to right and top to bottom.

  The function should fail without doing anything if the rectangle specified by `x`, `y`, `w`, and `h` falls outside the print bed, or if the current Z position does not allow for printing.

Any function with an `int` return type should return 0 for success and nonzero if there is any error.

Note: since sending a command and moving the print nozzle is very slow compared to CPU processing, your driver should try to minimize the number of instructions needed for each of these steps. For instance, it is very inefficient to print one unit of material at a time when you could print an entire row with one command.

Hardware API (provided)

The TipTap hardware provides a single function that your driver will use to issue low-level commands to the device:

```
int tapctl(uint32_t insn, void *data);
```

This function accepts a TipTap instruction `insn` and a pointer to a buffer that may be read from or written to, depending on the requested operation. The return value from `tapctl` is 0 if the operation executes successfully or a nonzero error code if there is an error. (You may pass this error code to the `tt_error` function to print the corresponding error message.) The function will display an error and abort if the virtual printer does something that would be a problem on a real-life 3D printer, such as printing without any supporting material underneath, printing where there is existing material, or printing without space between the nozzle and bed. This should not happen in any of the provided testcases.

The structure of a 32-bit TipTap instruction is shown in Table 1:

Bits	Field	Description
31–27	Opcode	Opcode of the command to be executed
26–23	Material	ID of material to use if printing
22–14	Distance	Signed distance to move (two's complement)
13–10	Flags	Flags that affect the operation of commands
9–0	SN	Sequence number of this instruction

Table 1: TipTap instruction format

You will use bitwise operators to pack these fields into a 32-bit integer to send to the hardware. (Tip: making a helper function to call `tapctl` is likely the best place to start the assignment.)

Opcodes

The following values may be used for the opcode field in a TipTap instruction. Each opcode uses a different set of fields; any fields which are not needed should be set to zero.

- `TT_POWERON`: Powers on the TipTap hardware and retrieves information about the size of the print bed. This command must be issued before any other TipTap instructions will be recognized. The `data` parameter should point to a buffer which will receive two 16-bit integers from the hardware: first the width (X dimension) of the print bed, then its height (Y dimension). When this command is issued, the print nozzle is reset to its home position of $(0, 0, d)$, where d is the depth (Z dimension) of the print bed. (Tip: to find the depth of the print area, use the laser probe at this point!)

- `TT_POWEROFF`: Powers off the TipTap hardware. This should be the last command issued when we are finished with the device.

- `TT_MOVEX`, `TT_MOVEY`, or `TT_MOVEZ`: Moves the print nozzle along the X, Y, or Z axis, respectively. The distance field in the instruction dictates how many units the nozzle is to move, and the sign conventions are illustrated in Figure 1: negative means left, up, or toward the print bed, and positive means right, down, or away from the print bed. You will need to keep track of the nozzle's location to avoid hitting the edges of the print space.

  To print material or obtain readings from the laser probe, issue one of these three commands with the desired flag set in the flags field of the instruction (see the next section).

Flags

Additional behavior can be requested by setting one of the flags in the instruction:

- `TTFLAG_EXTRUDE`: Directs the TipTap to extrude material as it moves. When using this flag, the material field in the instruction should be set to a nonzero number to indicate which material to use. The material will be extruded directly below the nozzle, such that if the nozzle is located at (x, y, z), then the printed material will be at $(x, y, z - 1)$.

  The `data` parameter should point to an array of bytes, with *distance* elements, indicating at which positions along the path material should be extruded. The printer will read one value from this array, print a unit of material if it is nonzero, move one unit, and repeat. For example, if the nozzle is at $(8, 0, 5)$, and the command `TT_MOVEX` is issued with the flag `TTFLAG_EXTRUDE`, distance -5, and `data` pointing to the array $\{1, 1, 0, 1, 0\}$, then one unit of the specified material will be printed to locations $(8, 0, 4)$, $(7, 0, 4)$, and $(5, 0, 4)$, with the nozzle ending at location $(3, 0, 5)$. This allows you to print several bits of one material in a single command, even if there are gaps or other materials in between.

  As a special case, when the distance field is 0, this flag will print one unit of material beneath the nozzle without moving it.

- `TTFLAG_PROBE`: Directs the TipTap to take readings using the laser probe as it moves. The `data` parameter should point to an array of *distance* unsigned 16-bit integers into which the reading at each location will be read. The reading from the probe is the amount of space between the nozzle and the nearest material directly beneath it (or the bed if there is none). For example, if the nozzle is at $(8, 0, 5)$ and there is material at $(8, 0, 2)$, the reading will be 2. If there were material directly beneath the nozzle at $(8, 0, 4)$, the reading would be 0. If there were no material beneath the nozzle, the reading would be 5.

The elements in the output array correspond in the same way as with `TTFLAG_EXTRUDE`: element 0 is the reading with the nozzle at its starting location, element 1 is after it has moved one unit, etc. The same special case applies, in that when the distance field is 0, this flag will perform a single reading without moving the nozzle.

It is an error to use both this flag and `TTFLAG_EXTRUDE` in the same command.

The other two flags are reserved and should be set to zero.

Sequence number

Each instruction is submitted with a sequence number, which should start at zero for the `TT_POWERON` instruction and increment by one for each instruction that is successfully executed. If an instruction has to be retransmitted, it should use the same sequence number. Since this value is 10 bits wide, it should wrap around to 0 after it reaches $2^{10} - 1$.

Testing

The provided simulator program will set up a virtual TipTap, 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 TipTap.
  -h         show this help
  -v         output verbose log data to stdout
  -o FILE    output log data to FILE

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

Workload files are provided in the `test` directory with the source code to test driver functionality, along with a script called `verify` which will run each of the workloads in turn. 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` pager:

```
./simulate -v < LOGFILE | less
```

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 -o LOGFILE < WORKLOADFILE
```

These options can also be passed to `verify` with the same effect.

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-s15/docs/assign3.tar.gz
```

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

- `driver.c`: Driver implementation. This is where you should write your code.
- `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!
- `tiptap.h`: Header file defining constants and functions needed by the hardware.
- `libtiptap.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 `libtiptap32.a` instead.)
- `simulate.c`: Program which reads a workload file of display commands from standard input and calls your implementation to handle them.
- `test/`: Directory containing a number of unit tests to help you find errors in your code.
- `verify`: Shell script to run all of the unit tests in the `test/` directory and verify their behavior and results.
- `Makefile`: Makefile for the entire project.

3. Comment your code as you write it to explain how everything works.

4. Implement the five device driver functions described in the "Device driver API" section. These functions are defined in the interface header file `driver.h`, and your implementations should go in `driver.c`. You may create helper functions in `driver.c` as needed.

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-uid1234-assign3.tar.gz`, where `LASTNAME` is your last name in all capital letters and `uid1234` 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-uid1234-assign3.tar.gz
```

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

```
$ mkdir /tmp/assign3test
$ cp LASTNAME-uid1234-assign3.tar.gz /tmp/assign3test
$ cd /tmp/assign3test
$ tar -xvzf LASTNAME-uid1234-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 handled according to the policy on the course website.

6. Any incorrect tarball (empty or containing something other than your completed code for this assignment) will be considered as not being submitted at all. We will try to notify you if we notice something is amiss, but it is up to you to confirm that your tarball is good by testing it as described above.

Bonus Points

Implement the `tiptap_scan(tt, x, y, w, h, output)` function. This function should use the laser probe to read the height of the printed material for the `w` × `h` rectangle starting at (`x`, `y`). The values should be stored into the `output` array. Note that this is the height of the printed material, not the distance as reported by the probe, so, for example, if there is no printed material at a certain coordinate, the value should be 0.

Important reminder

As with all assignments in this class, you are not permitted to:

- Copy any content from the Internet.
- Discuss or share ideas, code, configuration, text, or anything else related to the assignment with others in the class.
- Solicit help from anyone inside or outside the class other than the instructor, TAs, or assistants explicitly permitted by the instructor.

You may be asked at random by the instructor or TA to explain a portion of your code. If you cannot adequately explain how or why your own submission works, it will be considered highly suspect.

You are also responsible to protect your own ideas and 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 online, in person, etc., or use code from a source such as previous projects or other courses' lecture notes, please *document it in your submission.* This will go a long way!