ECE391: Computer Systems Engineering **Spring 2019**
Machine Problem 2

The Tux Controller

Introduction:

This machine problem gives you several opportunities to work with software that interacts directly with devices and must adhere to the protocols specified by those devices. Similar problems arise when one must meet software interface specifications, but you need experience with both in order to recognize the similarities and differences. The Tux Controller is that funny little game controller attached to each of the machines in the lab. It was designed and implemented by Kevin Bassett and Mark Murphy, two students who took this class during the 2004-2005 school year, and went on to work as lab assistants for the class.

The Tux Controller connects to the USB port of the lab machine. An FTDI "Virtual Com Port" (VCP) driver makes the USB port appear to software as a standard (old fashioned) RS232 serial port. We can then set up QEMU so that one of the emulated serial ports on the virtual machine maps to the emulated serial port connected to the Tux Controller.

Your tasks for checkpoint 2:

1. Enable the device driver to control the LEDs on the Tux controller
2. Enable the device driver to report when buttons are pressed and released
3. Handle device resets correctly
4. Enable control of the game using the Tux controller (support keyboard input simultaneously).
5. In the game, enable the up/down/left/right buttons on the Tux controller to do what the arrow keys do on the keyboard.

Remember, the Tux driver is expected to work with ANY user-level program. This means you will not receive full credit if you do not implement all `ioctl`s described in this document even if the game does not make use of them.

Set up instructions:

1. Remember to use Git or save a current version of your code as a backup.
2. You no longer need to compile the entire linux kernel, so you no longer need to boot your compiled kernel. To prevent your compiled kernel from running, right click on "test_debug.lnk" shortcut, go to properties and remove the -kernel ...
   bzImage from the target line.
3. Enable your test machine to make use of the serial port (named COM#) corresponding to the Tux controller. To check which COM number the Tux controller is currently connected on: Go to Start -> Devices and Printers -> right-click on FT232R USB UART -> Hardware tab -> USB Serial Port (COM#) where # should be less than 10. If it is higher, call a TA because it will not work with QEMU. Right click on "test_debug.lnk" shortcut, go to properties and add `-serial COM#` to the end of the target line. This connects the Windows COM# port to the Linux ttyS0 serial port. Note: Only your test machine will be able to communicate with the Tux controller with this setup. Although you can also add this option to your devel machine, you can only have one virtual machine use the Tux at a time. Therefore, we recommend you use the devel machine to compile your code and the test machine to load the kernel module and run the game.
4. Also remove the `-S` option from the end of the target line This option told `qemu` to wait until `gdb` was attached. In this assignment we don't need to start gdb until after the kernel is up and running.

5. When you first start your debug machine, a dialog will appear asking you for COM# Properties. The default values (Bits per second should be = 9600, Data bits = 8, Parity = None, Stop bits = 1, Flow control = None) will be correct. Click OK to continue booting your debug machine.

6. If there is no activity for approximately 2 hours, the Tux controllers will enter a low-power state and turn off the LEDs. This is intended to prolong the life of the LEDs. Pressing the RESET button will return it to a responsive state. Be aware of this if you sit down at a new machine or do not use the controller for more than two hours.

7. There is a bug in QEMU that may cause the test machine to freeze occasionally while using the Tux controllers. When it is frozen, you will be unable to connect to or interact with it from GDB. If that happens, go to the QEMU console (ctrl+alt+2), type `i 0x3f8` (this will pull bytes from UART like `in`), press the up arrow to get the command again and repeatedly enter it several times. Go back to the virtual terminal you were in (ctrl+alt+1). If it is still locked, repeat the steps above until it becomes responsive.

Building the Driver:

The `mp2/module` directory contains a framework for the driver you will be implementing for the Tux controller. A module is similar to a dynamically-loaded user-level library but instead of being used by an application it is used by the kernel to extend functionality. Modules allow for drivers to be loaded without having to recompile (or even reboot) the whole kernel. This simplifies the process of debugging as well. Modules can also be unloaded and a new version loaded into the kernel assuming that nothing catastrophic happened that would crash the kernel. Your driver code, of course, is running in the kernel, and bugs that are severe enough to crash the kernel are not impossible to produce.

The `module` directory uses a separate Makefile to compile a device driver module for use with Tux controller, thus building a module is similar to building a user level program: Change directories into the `module` directory and type `make`.

To **load** the module into the kernel, type `sudo /sbin/insmod ./tuxctl.ko` while in the module directory. A line should print out saying the tuxctl line discipline has been registered. You can safely ignore the "`module license 'unspecified' taints the kernel`" message. [1]

If you want to **remove** the module, issue the `sudo /sbin/rmmod tuxctl` command. This removes the module from kernel space and should print a line stating the tuxctl line discipline was removed. You can use these commands to install and remove the kernel module repeatedly during your development. If your module corrupts the kernel, however, you may eventually crash the system and have to reboot. If all else fails, try rebooting the test machine and then loading your module to see if it is really your latest code that is failing.

The module we have given you implements a tty line discipline. A line discipline is a driver that receives commands from the tty or serial port driver. In other words, it acts as a middle man between the serial port and the code you will implement to actually interpret the Tux controller commands. The code you will be implementing is in the `tuxctl-ioctl.c` file. You may add anything you deem necessary to the header files to make your implementation work. We also recommend you look at the mtcp.h file for a description of how the Tux controller works and some predefined constants, and at the appendix at the end of this document.

Writing Your Driver:

You will write portions of both the device driver which communicates with the Tux controller and the user-level application code to make use of the driver. On the driver side, you will need to add constants and definitions for the abstracted bit format to the header file. These constants (located in tuxctl-ioctl.h) will shared by the kernel and user-level code, ensuring that no inconsistencies arise. *There is no user-level test harness for MP2.* We advise you to write a simple test program to test your driver outside the game. We suggest that you write a test program to test the basic functionality (open, close, ioctls) before trying to make the game use the driver. We have provided input.c as a starting point for your user-level testing. To compile input.c type `make input`. We will test your driver with our own testing code, so be sure to test your driver's functionality thoroughly and adhering strictly to the spec. For the checkpoint, you must add the Tux controller functions (open, ioctls, close) to the input control in `input.c` and test it by compiling the file by itself. You will need to include `tuxctl-ioctl.h`. Only once you have debugged your input

[1] This simply indicates that the module you have loaded has not been designated as GPL and therefore taints the kernel so don't try to submit any bug reports to Linus.

control completely as a stand alone program should you try it with the full game. *To interface with the game you will need to create a new thread to take input from the Tux controller (you must support keyboard input simultaneously.* Opening the Tux controller is fairly simple. You may use the following code as an example to open the serial port and set the Tux controller line discipline:

```
fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY);
int ldisc_num = N_MOUSE;
ioctl(fd, TIOCSETD, &ldisc_num);
```

This code simply opens the tty port and sets the line discipline. You will want to add some error checking and whatever else you see fit to make this work with your code.

What Your Driver Has To Do:

Your device driver transforms the protocol supported by the Tux controller into a simpler abstraction for use by user-level programs such as the game. You must implement the following `ioctl`s for your device abstraction:

`TUX_INIT` Takes no arguments. Initializes any variables associated with the driver and returns 0. Assume that any user-level code that interacts with your device will call this `ioctl` before any others.

`TUX_SET_LED` The argument is a 32-bit integer of the following form: The low 16-bits specify a number whose hexadecimal value is to be displayed on the 7-segment displays. The low 4 bits of the third byte specifies which LED's should be turned on. The low 4 bits of the highest byte (bits 27:24) specify whether the corresponding decimal points should be turned on. This `ioctl` should return 0.

`TUX_BUTTONS` Takes a pointer to a 32-bit integer. Returns -EINVAL error if this pointer is not valid. Otherwise, sets the bits of the low byte corresponding to the currently pressed buttons as shown:

7							0
right	left	down	up	c	b	a	start

For full credit, use an interrupt-driven approach rather than polling.

Your changes to the device driver should be limited to the files `tuxctl-ioctl.h` and `tuxctl-ioctl.c`. The function `tuxctl_handle_packet` handles all packets received by the computer from the Tux controller. The function `tuxctl_ioctl` handles calls from user code (the game) to `ioctl`. You may add more `ioctl` calls, however the required `ioctl`s above must work as intended.

Tux Controller communication protocol:

The Tux controller protocol for interacting with the PC is described in `mtcp.h`, as you saw in PS2. Messages sent to the Tux controller are of variable length, while messages sent to the computer are always three bytes in length of the following general form:

Responses from the controller to the PC are always sent as three-byte packets of the following general form:

7							0	
0	1	R4	R3	0	R2	R1	R0	0
1	7 data bits							1
1	7 data bits							2

The 5-bit vector `R[4:0]` in byte 0 represents an opcode. All possible opcodes are defined in `mtcp.h` The other three bits in byte zero are fixed as indicated. Notice that bit 7 of each byte is fixed and used to frame the packets. This is a feature of the packet format because the COM port emulation in the old Microsoft Virtual PC software that we used to use was unreliable and occasionally lost bytes sent through it. The provided kernel patch includes code to detect this behavior and avoids providing you with broken packets. See the code in `tuxctl-ld.c` for details.

The Tux controller supports many different commands, each of which is defined in `mtcp.h`. You are only required to use a couple of them for this MP.

We recommend that you make use of `MTCP_BIOC_ON` and `MTCP_LED_SET`, in which case you must handle `MTCP_ACK` and `MTCP_BIOC_EVENT`. You must also handle `MTCP_RESET` packets sent to the computer by attempting to restore the controller state (value displayed on LEDs, button interrupt generation, *etc.*).

`MTCP_POLL` This command consists of a single `MTCP_POLL` byte (0xC2) sent to the controller: The response is a three-byte packet with the following format:

7							0	
0	1	0	0	0	1	0	0	0
1	X	X	X	C	B	A	S	1
1	X	X	X	R	D	L	U	2

That is, the first byte will have the value `MTCP_POLL_OK`, the second and third will contain active-low bit masks of which buttons are pressed; that is, a bit will be '0' when the corresponding button is pressed. The `C`, `B`, `A`, and `START` (S) buttons are in the second byte, and the `right` (R), `down` (D), `left` (L), and `up` (U) bits are in the third. Bits marked as 'X' in the diagram are unused.

`MTCP_LED_SET` See mtcp.h for the definition of the LED_SET bytes

Finally, a few important notes about the Tux controller:

For debugging purposes, when the controller receives an erroneous command, it locks up and displays `00P5` on the seven-segment display. Pressing the `reset` button returns it to a usable state. This behavior can be disabled by sending an `MTCP_DBG_OFF` command; after this is sent, an erroneous command will elicit an `MTCP_ERROR` response.

Additionally, when the controller is reset, for example by pressing the `reset` button, it asynchronously sends an `MTCP_RESET` packet to the PC. **Your code must detect this condition and re-initialize the controller to the same state it was in before the reset.** In order to accomplish this your code must internally keep track of the state of the device.

Debugging the Tux Controller:

Debugging in kernel space is not as simple as debugging in user space. You made find it helpful to use `printk()` from your driver module. `printk()` is like `printf()` for kernel mode.

For when the kernel loads the tuxctl module, it may load it anywhere it pleases in its address space. This means GDB needs to know where the module was loaded before it can start debugging it. The object file for the tuxctl also needs to be loaded into GDB so that the proper symbols can be resolved.

Once your debug machine loads, load the tuxctl module (see above). You now need to find out where in memory the kernel loaded the module. Issue the command `cat /proc/modules` to print out the location of all modules currently loaded in the kernel. You should see a line such as

```
tuxctl 6792 0 - Live 0xd0813000 (P)
```

at the top. `0xd0813000` is the address the module was loaded at. (It will probably be different for you). Back in your devel machine, in the `module` directory, launch gdb with no arguments. Once in GDB, type `add-symbol-file ./tuxctl.o ADDRESS` where ADDRESS is the address that was printed out in for the module in your debug machine, 0xd0813000 in my case. GDB will then ask you to confirm. Type `y` and hit `enter`. Now you will need to connect GDB to the remote machine by typing `target remote 10.0.2.2:1234`. You may now define breakpoints for your driver and continue as normal.

Again, make use of the stand-alone input program in `input.c` before tackling the full game.