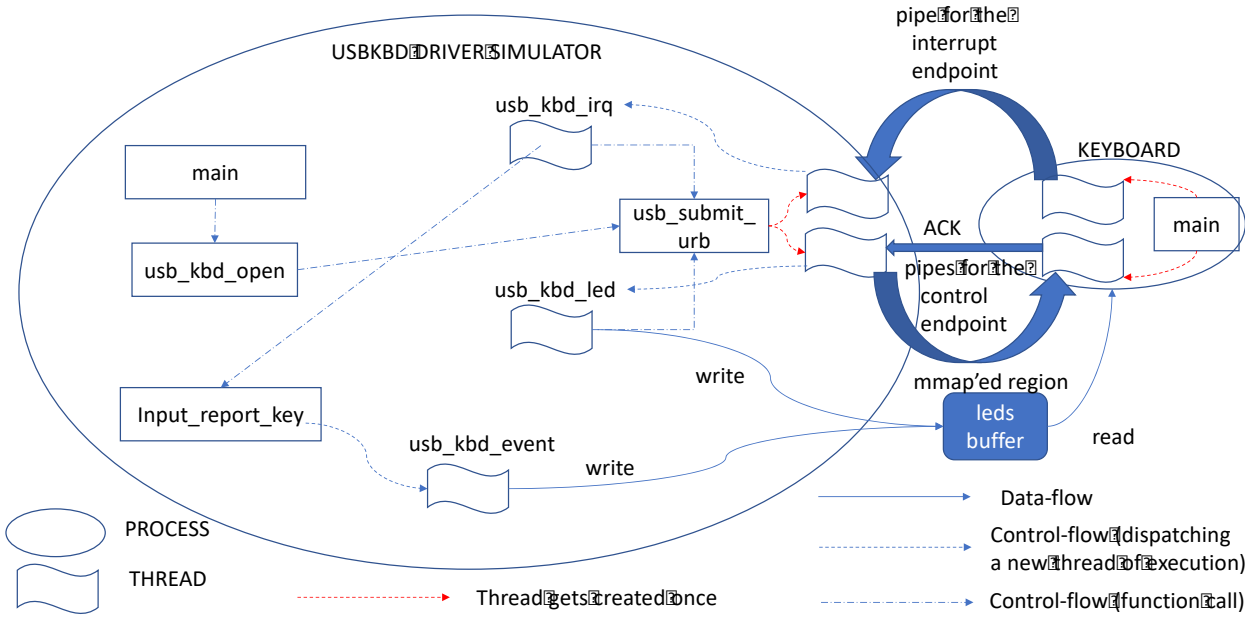
EEL 4732/5733 Advanced Systems Programming
Assignment 7.a

In this assignment, you are going to use and demonstrate your understanding of the usbkbd driver with respect to the concurrency aspect of its I/O communication (Please see the slides USBKeyboard.pptx on CANVAS on a detailed explanation of the operation of the driver). You will implement a **user space simulator** that closely models the callbacks that are defined as the URB completion handlers (usb_kbd_irq and usb_kbd_led) as well as the callback functions that get registered with the Input subsystem: usb_kbd_open and usb_kbd_event. You should also simulate the communication between the USB core and the keyboard device by simulating the nature of the communication with each endpoint (the interrupt endpoint and the control endpoint) and through different channels. If your simulator is close to the original implementation of the usbkbd driver, you should be able to observe correct functioning of the keyboard: No key events are lost and LED status on the keyboard are consistent with the key presses.

ASSIGNMENT 7: THE ARCHITECTURE OF THE SIMULATOR FOR THE USBKBD DRIVER



You are going to use the architecture shown in the figure to implement your simulator as a user space application. Specifically:

1. Your simulator will be a multi-process and a multi-threaded application.
2. The custom data structure (struct usb_kbd) of the usbkbd driver will be simulated by including all the relevant parts. You should also implement the struct input_dev type by including the event callback function pointer field and the led field.

3. The application will start by calling the usb_kbd_open function from the main function.
4. Each execution of each URB completion handler will be implemented by creating a new thread. Similarly, each execution of the usb_kbd_event function will be in a new thread, which will be created by the input_report_key function when there is CAPSLOCK event.
5. The keyboard device and the driver will be implemented as separate processes. The driver process will also include some API of the USB Core and the input subsystem.
6. USB Core will communicate with different endpoints of the keyboard device via the pipe mechanism (using the pipe, read, and write system calls). Each communication will be handled by a different dedicated thread (created once), each of which will be created the first time usb_submit_urb gets called for a given URB (irq or led) and will keep existing until the end of the simulation.
7. The input to the simulator is a text file including the sequence of key events. The keyboard process will write each character in the file to the pipe (see below for the meaning of the characters).
8. The keyboard process will receive a generic control command one pipe, read the leds buffer, and carry the LED related command based on the content of the leds buffer, and respond with an ACK on the other pipe.
9. Your simulator should implement the following mechanisms to implement various components:
    a. Communication with each endpoint will be achieved using a pipe by using the pipe system call. So, you will create a separate pipe for each endpoint. To simplify things, one pipe will be involved for the interrupt endpoint and two pipes for the control endpoint as shown in the figure. The interrupt endpoint will be simulated by the write end of the relevant pipe and the control endpoint will be simulated by the read end of the relevant pipe.
    b. The keyboard will be implemented as a process. It will take a text file that represents the key events and write it to the pipe. USB Core thread will read one character at a time to simulate a window of size 1 byte through the INTERRUPT endpoint. Alphanumeric characters will represent key press events of such characters (we will not model key releases in this assignment except for the CAPSLOCK key). The following characters will have special meaning:
        i. #: no key event
        ii. @: CAPSLOCK press
        iii. &: CAPSLOCK release

        USB Core will simulate continuous polling by reading one character (byte) at a time in a loop until it reads a character other than the # character and will execute usb_kbd_irq function in a new thread.

    The keyboard process will get a control command denoted by C from the pipe connected to its CONTROL endpoint and will read the actual command from the leds buffer that will be stored in an mmapped region:
        iv. 1: Change the status of the CAPSLOCK LED (if on, turn it off. If off, turn it on).

v. 0: Do not change the status of the CAPSLOCK LED (if on, keep it on. If off, keep it off).

10. For the input layer, you will implement the input_report_key function, which will record if there is a CAPSLOCK press event in kbd->dev->led. It will execute the usb_kbd_event function in a separate thread if the event is a CAPSLOCK press or a CAPSLOC release, where the former will result in writing a value of 1 and the latter will result in a value of 0 to the leds buffer.
11. The usbkbd simulator should print the characters pressed on the standard out in lower case or upper case based on the CAPSLOCK presses. You should assume that initially we are in lower case mode. The keyboard should produce the very last output on the standard output and print all sequences of LED actions (CAPSLOCK turn on or turn off).
12. Use the Pthreads library and the relevant system calls to implement the simulator. To get full credit, you will need to follow the architecture shown in the figure. Submit the source code of your simulator, a Makefile, and a README file.
13. You can use the following test cases to test your code:
    a. Input file content: Hello#@& world@&# every@&#one###!
       Output on standard output:
       Hello WORLD everyONE!
       ON OFF ON
    b. Input file content: @&@&Hello
       Output on standard output:
       Hello
       ON OFF