ECE391: Computer Systems Engineering **Fall 2021**
Problem Set 2 **Due: in Gitlab repository by 5:59PM CST Tuesday 21 September**

Work in groups of at least four people. The person submitting the code **(and NO ONE ELSE IN THE GROUP)** should list all group members' netids, including their own, in the file `partners.txt`, with each netid on a separate line. For example, `partners.txt` should contain the following if submitted by `yanmiao2`:

```
yanmiao2
xiangl5
hc19
sjeon12
ch5
```

WARNING: Any `partners.txt` files that conflict with any other `partners.txt` file will earn 0 pts for all concerned.

Problem 1 (6 points): Reading Device Documentation

As preparation for MP2, you need to figure out how to do a couple of things with the VGA. There's some free documentation available at `http://www.osdever.net/FreeVGA/vga/vga.htm`, but you may use any documentation that you like to learn the necessary changes to registers and so forth.

a. You must use VGA support to add a non-scrolling status bar. Figure out how to use VGA registers to separate the screen into two distinct regions. Explain the necessary register value settings, how the VGA acts upon them, and any relevant constraints that must be obeyed when setting up the status bar.

b. You must change the VGA's color palette. Figure out how to do so, and explain the sequence of register operations necessary to set a given color to a given 18-bit RGB (red, green, blue) value.

Write your answers in 'p1/solution.md'.

Problem 2 (9 points): Documentation in Files

As part of MP2, you will also write a device driver for the Tux controller boards in the lab. The documentation for this board can be found in the file `mtcp.h` which is available on the course website. You will need to read it for the following questions.

a. For each of the following messages sent from the computer to the Tux controller, briefly explain when it should be sent, what effect it has on the device, and what message or messages are returned to the computer as a result: `MTCP_BIOC_ON`, `MTCP_LED_SET`.

b. For each of the following messages sent from the Tux controller to the computer, briefly explain when the device sends the message and what information is conveyed by the message: `MTCP_ACK`, `MTCP_BIOC_EVENT`, `MTCP_RESET`.

c. Now read the function header for `tuxctl_handle_packet` in `tuxctl-ioctl.c`—you will have to follow the pointer there to answer the question, too. In some cases, you may want to send a message to the Tux controller in response to a message just received by the computer (using `tuxctl_ldisc_put`). However, if the output buffer for the device is full, you cannot do so immediately. Nor can the code (executing in `tuxctl_handle_packet`) wait (for example, go to sleep). Explain in one sentence why the code cannot wait.

Write your answers in 'p2/solution.md'.

Problem 3 (18 points): Martian Language Translator

In the year 2031, we are scheduled to launch Starship X-3000 and send Melon Busk to Mars for preliminary research on the local customs of the Martian people. The Yellow Origin company is creating Martian-English translation software so that Melon can talk with the local dwellers. The device on which the translating program will run has multiple restrictions imposed by the physical environment on Mars. As a summer intern at Yellow Origin, you are tasked with implementing the synchronization mechanism of the translator program in the form of a C library for use within the Linux kernel. In this translator program, Melon and the Martian can input a message in their own language, or pick up the translation result of what has been input by the other entity.

- You are provided with a set of four low-level routines that interface directly to the translation and queueing hardware. The message queues are maintained by the translator core, so you only need to invoke the given routines (translate_to_english, get_translation_in_martian, and so forth) to add/get a message to/from the translator queues. However, since the hardware has yet to be fully implemented, your code must be conservative and ensure that none of these routines is ever allowed to execute simultaneously with another (nor with itself).
- The routines provided also do nothing to protect against queue overflow nor underflow, so your code must also ensure that message loss through overflow does not occur (nor receipt of non-existent/duplicate messages through underflow). Due to the hardware limitations, the **queues of translated messages** hold at most 4 messages in Martian Language and 10 messages in English. The queues do not share storage resources, so it is not possible to store more of one type of message when fewer of the other type are enqueued.
- Assume that one English message always translates to one Commom Martian Language message and vice-versa.
- You must produce a set of four thread-safe routines—two for input and two for output—that support use of the translation system. Your routines will be called by multiple threads within the Linux kernel. **Do not make assumptions about whether the calls are made within system calls or from interrupt context.**
- Melon and the Martian friend both have multiple threads to input/retrieve content.
- When Melon or the Martian friend wants to speak, one of their many threads calls one of your input routines. If the appropriate output queue is full, your routine should wait indefinitely until space is available. To better learn the Martian customs, Melon should speak less and listen more. Therefore, if a message sent by the Martian friend is available when Melon tries to provide input, your routine should delay translation of Melon's message until his threads have emptied the queue of messages from the Martian (another thread or threads will have to do so). Once all conditions have been met, your input routine must translate the message and enqueue it.
- There is no need to enforce order among messages sent by the same person. In other words, a message can be translated and enter the queue before a message that arrived earlier.
- For full credit, your routines must not repeatedly acquire and release locks when no changes have been made to the relevant queue status.
- You may ignore the possibility of deadlock that can occur if all processors are blocked waiting to enqueue new messages.
- When Melon or the Martian friend are ready to listen, one of their many threads calls one of your output routines. If the appropriate queue is empty when this happens, your routine should return -1. Otherwise, your routine should copy the message into the buffer and return 0.

Fill in the code for all four functions provided in ‘p3/solution.c’. An excerpt of the file is provided here. Remember that these functions may be called simultaneously on multiple processors from system call and/or interrupt contexts. You may use only one `spinlock_t` in your design. No other synchronization primitives may be used.

```
typedef struct martian_english_message_lock {
    // The type of synchronization primitive you may use is spinlock_t.
    // You may add up to 3 elements to this struct.



} me_lock;

void melon_input(me_lock* lock, msg* message) {







}

void martian_input(me_lock* lock, msg* message) {







}

int melon_get_output(me_lock* lock, msg* message) {







}

int martian_get_output(me_lock* lock, msg* message) {







}
```

Use these four routines to interface to the translation hardware and queueing system. Note that the translate_to routines do not check for full output queues, nor do the get_transation routines check for empty queues. **None of the four routines should be called simultaneously with any others (including themselves).**

```
// You do not need to define these functions, but you need to call them in
// your synchronization interface.
/*
 * translate_to_english
 *   DESCRIPTION: translate the message in the buffer to English;
 *      then put the message in the corresponding queue.
 *   INPUTS: message - a pointer to the input message buffer.
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECT: none
 */
void translate_to_english(msg* message);

/*
 * translate_to_martian
 *   DESCRIPTION: translate the message in the buffer to Martian;
 *      then put the message in the corresponding queue.
 *   INPUTS: message - a pointer to the input message buffer.
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECT: none
 */
void translate_to_martian(msg* message);

/*
 * get_translation_in_english
 *   DESCRIPTION: get a translated message in English.
 *   INPUTS: message - a pointer to the message buffer which will
 *      be filled in with a translated English message.
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECT: Will fill in the given message buffer.
 */
void get_translation_in_english(msg* message);

/*
 * get_translation_in_martian
 *   DESCRIPTION: get a translated message in Martian.
 *   INPUTS: message - a pointer to the message buffer which will
 *      be filled in with a translated Martian message.
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECT: Will fill in the given message buffer.
 */
void get_translation_in_martian(msg* message);
```