ECE391: Computer Systems Engineering **Spring 2022**
Spring 2022 **Problem Set 2**
Due: in Gitlab repository by 11:59PM CST Monday 14 February

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 `naiyinj2`:

```
naiyinj2
xiangl5
henryh2
yamsani2
```

Note: There is no autograder for this problem set.

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

Problem 1: Reading Device Documentation (4pts)

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.

Problem 2: Documentation in Files (6pts)

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` in the class directory under mp2. 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 (*e.g.*, go to sleep). Explain in one sentence why the code cannot wait.

Problem 3: Synchronization (18pts)

There is a laboratory which can be occupied by either zombies or scientists. Now you are hired to design a system that satisfies the following rules:

- At all times, the lab must either be empty, contain scientists, or contain zombies. At most 10 zombies or 4 scientists can stay in the lab, but there is NO limit for the number of zombies/scientists waiting in line.
- If there are scientists occupying the lab, zombies must wait for all scientists to leave before entering. Similarly, if zombies occupy the lab, scientists must wait for them to leave before they can enter.
- While waiting in line, scientists have higher priority. This means that if there are scientists waiting to enter, no zombie can enter the lab (even if the lab is currently occupied by zombies). You can think of this as scientists always being moved to the front of the line. Note that when a scientist joins the line and the lab is occupied by zombies, the scientist still must wait for the zombies in the lab to leave; scientists cannot force zombies out of the lab.
- There is no need to enforce priority among zombies or among scientists (ie. if A arrives before B, A does not necessarily enter the lab before B)
- When zombies occupy the lab, they introduce contamination. To avoid contaminating the scientists' work, the lab must be sanitized before scientists can enter a lab previously occupied by zombies. No sanitization need be done when switching from scientists to zombies. Sanitization must be done as infrequently as possible, and must only be done when the lab is empty.

You are to implement a thread safe synchronization library to enforce the lab occupancy policy described above.

A set of **_enter** functions will be called by scientists and zombies to enter the lab. These functions should behave like a "lock," and do not return until the lab has been successfully entered by the caller. If any errors occur, return -1. Otherwise, return 0.

When a scientist or zombie wants to leave the lab, they call their **_leave** function to let your library know they have left.

You may use only one spinlock in your design, and no other synchronization primitives may be used.

As these functions will be part of a thread safe library, they may be called simultaneously

Write the code for enter and exit of scientist/zombie. Assume that these functions may be called simultaneously from multiple threads.

```
// You may add up to 5 elements to this struct.
// The type of synchronization primitive you may use is SpinLock.
typedef struct zs_enter_exit_lock{

}zs_lock;


int zombie_enter(zs_lock* zs) {







}

int zombie_exit(zs_lock* zs) {







}

int scientist_enter(zs_lock* zs) {







}

int scientist_exit(zs_lock* zs) {







}

// You need not define this function, but you must call it in
// your synchronization interface..
/*
 * sanitize_lab
 *   DESCRIPTION: Sanitizes the lab, removing all zombie contaminants.
 *   INPUTS: zs -- pointer to the zombie-scientist lock structure
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECTS: Must only be called when the lab is empty.
 */
void sanitize_lab(zs_lock* zs);
```