ECE391: Computer Systems Engineering **Spring 2018**
Machine Problem 1 **Due: Feb 5, 6:00pm**

Text-Mode Fish Animation

Demo (Feb 5, Monday, 6-8:30pm): Last name starts with A-L
Demo (Feb 6, Tuesday, 6-8:30pm): Last name starts with M-Z
Note: Please put your name in Chara, we will start a new handin queue 15 minutes before your handing time slot.

In this machine problem, you will modify the Linux real-time clock (RTC) driver to toggle characters on the text-mode video console from one ASCII character to another, with a user-settable toggle rate. This will serve a dual purpose: first, it will be an exercise in writing x86 assembly, allowing you to gain experience with the x86 ISA. Second, it will provide an introduction into how drivers accomplish tasks inside the Linux kernel.

Please read the entire document before you begin.

A Note On This Handout: The sections entitled "Linux Device Driver Overview," "RTC Overview," "Ioctl Functions," and "Tasklets" contain background Linux knowledge which is not critical for you to complete this MP. The material described in these background sections will be covered in lecture in the next few weeks, but it may be helpful to read these sections to familiarize yourself with the context of your code in this MP.

MP1 Assignment

You will add four new `ioctl`s to the existing RTC driver, as well as a tasklet that will update the text-mode video screen on every RTC interrupt.

Your code will reside in `mp1.S`, a GNU-style assembly file. Assembly files with a capital-S extension (.S) are preprocessed using the standard C preprocessor before being assembled, so things like `#include` and `#define` are OK to use. Your code must be implemented using GNU x86 assembly.

Note: Do not use `#` to start a comment at start of line.

MP1 Data Structure

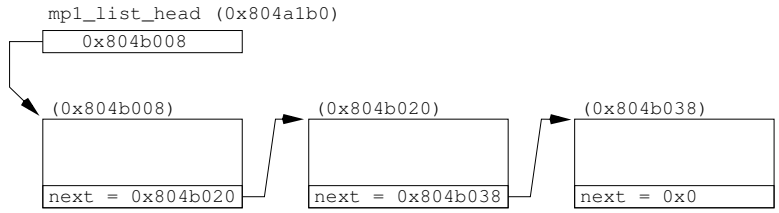
The main structure you will be working with is `mp1_blink_struct`.

```
struct mp1_blink_struct {
    unsigned short          location;   /* Linear offset on text-mode buffer */
    char                    on_char;    /* Char to put during "on" period */
    char                    off_char;   /* Char to put during "off" period */
    unsigned short          on_length;  /* Length of on period
                                         * in number of RTC interrupts */
    unsigned short          off_length; /* Length of off period */
    unsigned short          countdown;  /* Number of RTC interrupts left in period */
    unsigned short          status;     /* Status word (on=1/off=0) */
    struct mp1_blink_struct *next;      /* pointer to next item in linked list */
}
```

This structure definition is usable only in C programs. There are constants defined for you at the top of the provided `mp1.S` that give you easy access to the fields in this struct from your assembly code. See the comments in `mp1.S` for further information on how to use them.

To implement characters "blinking" on the text-mode video console, a linked list will be created by your modified RTC driver that will allow any location on the text-mode video console to be toggling characters from `off_char` to `on_char` and back, with toggle rates determined by `on_length` and `off_length`. A pointer to the first element in the linked list (the **head** of the list) is defined in the mp1.S file as a global variable, `mp1_list_head`. `mp1_list_head` is

initialized to NULL (the value it holds is zero) to indicate that there are currently no blinking locations on the screen. The tail element of the list will have its `next` field equal to NULL to indicate that it is the last element. A diagram of this singly-linked list layout for a three-item list is shown on the following page. Example memory addresses of structures and variables are shown in parentheses.



MP1 Tasklet

The first function you need to write is called `mp1_rtc_tasklet`. The tasklet must update the state of the game. Its C prototype is:

```
void mp1_rtc_tasklet (unsigned long);
```

Every time an RTC interrupt is generated, `mp1_rtc_tasklet` will be called. Your tasklet will walk down the `mp1_list_head` list, examining each `mp1_blink_struct` structure. The function first decrements the `countdown` field of the structure. If the `countdown` field has reached zero after the decrement, the tasklet will examine the `status` field. If this field is equal to 1, that location currently has the `on_char` character; if this field is 0, that location currently has the `off_char` character. The tasklet should put the opposite character (i.e. interchange the status between on/off) out to video memory with a call to `mp1_poke`. For information on how to draw to the screen, see the "Text-Mode Video" section. Finally, the tasklet updates the `countdown` field by copying the value from the opposite length field to `countdown`. For example, if the character was currently off and you just turned it on, copy `on_length` to `countdown`. In this way, the toggle rate for each character is controlled by the length fields. The tasklet then must move on to the next list element. The function returns when it reaches the end of the list.

MP1 Ioctls

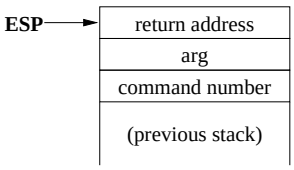
The next function you must write is called `mp1_ioctl`. Its C prototype is:

```
int mp1_ioctl (unsigned long arg, unsigned long cmd);
```

This function serves as a "dispatcher" function. It uses the `cmd` argument to determine which of the next four functions to jump to. The table below gives a brief summary of `cmd` values, the corresponding core function, and a brief description of what that core function does. Each of the core functions are described in the section entitled "Core Functions." Note that you must check this `cmd` value; if it is an invalid command, return `-1`.

cmd value	**Core function**	**Description**
0	`mp1_ioctl_add`	add a blinking location
1	`mp1_ioctl_remove`	remove a blinking location
2	`mp1_ioctl_find`	get information about a location
3	`mp1_ioctl_sync`	synchronize a new blink location with an existing one
other	-	Any value other than 0-3 is an error. Return `-1`.

The method used to jump to one of the core functions is to use **assembly linkage** without modifying the stack. A picture of the stack at the beginning of `mp1_ioctl` is shown below.



Each of the core functions takes `arg` directly as its parameter. Since this parameter is passed to the `mp1_ioctl` function as its first parameter, `mp1_ioctl` can simply jump directly to the starting point of one of the core functions without modifying the stack. The `arg` parameter will already be the first parameter on the stack, ready to be used by the core function. In this way, it will appear to the core functions as if they were called directly from the RTC driver using the standard C calling convention without the use of this assembly linkage. Your `mp1_ioctl` must use a jump table—see the section "Jump Tables" below.

Core Functions

You must implement each of the following four functions in assembly in the `mp1.S` file.

Note: A common task across these four `ioctl`s is searching a linked list for a specific element that matches a particular location. You must implement a separate function that performs a linked list search, and call this function from the `mp1_ioctl_remove`, `mp1_ioctl_find`, and `mp1_ioctl_sync` core functions. Designing the interface to this function (*i.e.*, what parameter(s) is/are passed to it, what value(s) is/are returned from it, *etc.*) is up to you.

`int mp1_ioctl_add(unsigned long arg)`
The add `ioctl` takes as its argument a user-level pointer to a `mp1_blink_struct` structure. First, dynamically allocate memory using the `mp1_malloc` function to store a copy of the structure. Copy the entire `mp1_blink_struct` from the user-space structure to the newly-allocated memory (use `mp1_copy_from_user`). Then set the `countdown` field to be equal to the `on_length` field, and set the `status` field to 1. Then insert this structure at the head of the linked list using the `mp1_list_head` pointer. Finally, make a call to `mp1_poke` with the correct register parameters to immediately display the character on the text-mode video screen. This effectively turns the location "on." After `countdown` RTC interrupts have elapsed, your `mp1_rtc_tasklet` will turn the location "off." This function should return 0 if a successful add was performed.

Your function must handle errors. If there is a memory allocation error (*i.e.*, `mp1_malloc` returns NULL), return `-1`. Remember the semantics of `mp1_copy_from_user`. If it could not copy all the bytes requested, it will return the number of bytes it was not able to copy. If this function returns anything other than 0, the copy has failed, and the function should return `-1`. If the `location` is outside the valid range of 0 to 80*25-1, this function should return `-1`. Finally, your error handling must prevent memory leaks. If you have allocated any memory using `mp1_malloc`, and you find that there is an error condition, you must free the memory using `mp1_free`.

`int mp1_ioctl_remove(unsigned long arg)`
The remove `ioctl` takes an integer location as its parameter. Traverse the `mp1_list_head` list, looking for an element whose `location` field matches the argument given to this function. If there is such an element, remove it from the linked list and free its memory with a call to `mp1_free`, and return `0`. If there is no element whose `location` matches, return `-1`.

`int mp1_ioctl_find(unsigned long arg)`
The find `ioctl` takes a pointer to a `mp1_blink_struct`, like add. The only parameter it is concerned with as an input is the location parameter, but you must validate that the pointer refers to a valid structure before reading from the structure. After extracting the location parameter from the user-level structure, search the `mp1_list_head` list for an element that matches the location. Then copy the entire element, which is a `mp1_blink_struct`, to the user-level structure pointed to by the parameter that was passed in (use `mp1_copy_to_user`). In this way it uses the parameter as both an input and an output. If there is no matching location in the list, return `-1`, otherwise return `0`. Similar error conditions apply to this function as in the previous two.

`int mp1_ioctl_sync(unsigned long arg)`
The sync `ioctl`'s `unsigned long` argument is really two two-byte `unsigned short` integers, packed into one four-byte argument. The first integer is stored in the upper 16 bits of `arg`, and the second integer is stored in the lower 16 bits. You must extract these two integers from the single argument.

The sync `ioctl` synchronizes two existing locations on the screen. The first integer represents the location of the first blinking character, and the second integer represents the location of the second blinking character that will become synchronized with the first. Search the `mp1_list_head` list, looking for elements with locations that match the two integers, respectively. Then copy the timing information (the `on_length`, `off_length`, `countdown`, and `status`

fields) from the first element to the second element. After copying these fields, call `mp1_poke` to immediately update the display using the correct character (*i.e.*, either the `on_char` or the `off_char`, depending on `status`) to the screen for the second location. This function should return `0` on success, and -1 on failure. Similar failure cases apply.

Synchronization Constraints

The code (both user-level and kernel) for MP1 allows the tasklet to execute in the middle of any of the `ioctl`s, so you must be careful to order the updates properly in some of the operations. Since the tasklet does not modify the list, the main constraint is that any `ioctl` that modifies the list does so in a way that never leaves the list in an unusable state.

In particular, `mp1_ioctl_add` must fill in the newly allocated structure, including the `next` field, *before* changing the head of the list to point to the new structure. Similarly, `mp1_ioctl_remove` must remove the element from the list before freeing it; copying the structure's `next` pointer into a register is not sufficient, since the tasklet could try to read the structure after the call to `mp1_free`. Updates in the other calls can not lead to major problems.

Suggested Order of Writing Functions

Below is our suggested order of writing functions. If you write the functions in this order, you can test them as you go along and the expected output should be what is described. This way, you can have some confidence in the portions of code you write instead of writing everything all at once and then testing it only to find out something does not work and you have to look through every single line of every single function to track the bug. Feel free to deviate from this list if it is more convenient. And of course, do write your own test cases that individually test the functions you have written (formally called unit testing).

1. IOCTL dispatcher - There won't be any output when this is finished. This is just to set up the calls to the core functions.
2. ADD - An ASCII picture of a fish should appear if this is working correctly
3. Tasklet - The ASCII fish should blink between the two frames
4. FIND/SYNC - An I/M should appear after a few seconds and the "I/M" blinks should sync up with the rest of the fish background after some time.
5. REMOVE - After the "I/M" have synced with the rest of the fish background, the "M" will be removed and so the blinking will stop and the "I" will be left over

Getting Started

Be sure that your development environment is set up from MP0. In particular, have the base Linux kernel compiled and running on your test machine.

Begin MP1 by following these steps:

- We have created a `Git` repository for you to use for this project. The repository is available at `https://gitlab.engr.illinois.edu/ece391_sp18/mp1_<YOUR_NETID>` and can be accessed from anywhere.
- Access to your Git repositories will be provisioned shortly after the MP is released. Watch your @illinois.edu email for an invitation from Gitlab.
- To use Git on a lab computer, you'll have to use `Git Bash` on Windows, not the VM. You are free to download other Git tools as you wish, but this documentation assumes you are using `Git Bash`. To launch `Git Bash`, click the `Start` button in `Windows`, type in `git bash`, then click on the search result that says `Git Bash`.
- Run the following commands to make sure the line endings are set to LF (Unix style):
  ```
  git config --global core.autocrlf input
  git config --global core.eol lf
  ```
- Switch the path in git-bash into your `Z:` drive by running the command: `cd /z`
- If you do NOT have a ssh-key configured, clone your git repo in `Z:` drive by running the command (it will prompt you for your NETID and AD password):
  ```
  git clone https://gitlab.engr.illinois.edu/ece391_sp18/mp1_<YOUR_NETID>.git mp1
  ```

  If you do have a ssh-key configured, clone your git repo in `Z:` drive by running the command:
  ```
  git clone git@gitlab.engr.illinois.edu:ece391_sp18/mp1_<YOUR_NETID>.git mp1
  ```

In your `devel` machine:

- Change directory to your MP1 working directory (`cd /workdir/mp1`). Inside there, you should find a file called `mp1.diff`. Copy this file to the Linux kernel directory with
  ```
  cp mp1.diff /workdir/source/linux-2.6.22.5
  ```
- Now change directory to the Linux kernel directory (`cd /workdir/source/linux-2.6.22.5`). Apply the `mp1.diff` patch using
  ```
  cat mp1.diff | patch -p1
  ```
  The last argument contains a digit 1, not the lowercase letter L. This command prints the contents of mp1.diff to `stdout`, and then pipes `stdout` to the `patch` program, which then applies the patch to the Linux source. You should see that the patch modified three files, `drivers/char/Makefile`, `drivers/char/rtc.c`, and `include/linux/rtc.h`. Do *NOT* try to re-apply the patch, even if it did not work. If it did not work, revert all 3 files to their original state using SVN (`svn revert <file name>`). After that, you may apply the patch again.
- Change directory back to `/workdir/mp1`. You are now ready to begin working on MP1.
- **Do not commit the linux source or the kernel build directory. The number of files makes checking out your code take a very long time. If during handin, we find the whole kernel source or the build directory in your repository, you will lose points.** We have added a .gitignore file to your initial repository. This file contains all the Git ignore rules that tells Git to not commit the specified file types. The linux source and kernel build directory are one such example of files that will be ignored. Try and explore the .gitignore file to see what other file types will be ignored.

Be sure to use your repository as you work on this MP. You can use it to copy your code from your development machine to the test machine, but it's also a good idea to commit occasionally so that you protect yourself from accidental loss. Preventable losses due to unfortunate events, including disk loss, will not be met with sympathy.

Testing

Due to the critical nature of writing kernel code, it is better to test and debug as much as possible outside the kernel. For example, let's say a new piece of code has a bug in it where it fails to check the validity of a pointer passed in to it before using it. Now, say a NULL pointer is passed in and the code attempts to dereference this NULL pointer. When running in user space, Linux catches this attempt to dereference an invalid memory location and sends a signal, [1] SEGV, to the program. The program then terminates harmlessly with a "Segmentation fault" error. However, if this same code were running inside the kernel space and an invalid pointer were dereferenced, the kernel would crash and the only recourse would be to restart the machine.

In addition, debugging kernel code requires the setup you developed in MP0—two machines, connected via a virtual TCP connection, with one running the test kernel and the other running a debugger. In user space, all that's necessary is a debugger. The development cycle (write-compile-test-debug) in user space is much faster.

For these reasons, we have developed a user-level test harness for you to test your implementation of the additional `ioctl`s and tasklet. This test harness compiles and runs your code as a user-level program, allowing for a much faster development cycle, as well as protecting your test machine from crashing. Using the user-level test harness, you can iron out most of the bugs in your code from user space before integrating them into the kernel's RTC driver. The functionality is identical to the functionality available if your code were running inside the kernel.

The current harness tests some of the functionality for all the ioctls, but it is not an exhaustive test. It is up to you to ensure that all the functionality works as specified, as your code will be graded with a complete set of tests.

Note: For this assignment, a test harness is provided to you that can test some of the functionality of your code prior to integration with the actual Linux kernel. Future assignments will place progressively more responsibility on you, the student, for developing test methods. What this means is that a complete test harness will not be provided for every MP, and it will be up to you to design and implement effective testing methods for your code. We encourage you to look over how the user-level test harness works for this MP, as its design may be of use to you in future MPs. This test harness is fully functional, and uses some advanced programming techniques to achieve a complete simulation of how your code will execute inside the Linux kernel. You need not understand all of these techniques; however, understanding the important ideas is useful. Questions on Piazza as to how this test harness works are welcome as well.

Running the user-level test program: To run the user-level test program, follow these steps:

- Type `cd /workdir/mp1` to change to your MP1 working directory.
- Type `make` to compile your code and the test harness.
- Type `su -c ./utest` to execute the user-level test program as `root` (you will need to type `root`'s password).

You can also type `su -c "gdb utest"` to run gdb on the user-level test harness to debug your code. Debugging the kernel code will be difficult. Use the `disas` (disassemble) command on `mp1_rtc_tasklet` or `mp1_ioctl` to see the start of your code (feel free to add more globally visible symbols), then use explicit addresses to see the rest of it. Be sure to start any disassembly with the starting byte of an instruction rather than an address in the middle of an instruction. With non-function symbols (such as those in your assembly code), and with addresses, you need an asterisk when identifying a breakpoint, *e.g.*, `break *mp1_ioctl` or `break *0x12345678`.

The test code changes the display location to the start of video memory. If you do not see a prompt after the code finishes (correctly or otherwise), pressing the Enter key will usually return the display to normal. Note also that `gdb` will return the display to its usual location, after which you will not be able to see any of the animation (while debugging).

Note: When running the user test under `gdb`, the debugger frequently stops your program whenever a signal (such as SIGUSR1 or SIGALRM) occurs. To turn off this behavior and make it easier to debug your program, type the following commands in `gdb`:

```
handle SIGUSR1 nostop noprint
handle SIGALRM nostop noprint
```

[1] Think of a signal as a user-level (unprivileged) interrupt for now.

Testing your code in the kernel: Once you are confident that your code is working, you need to build it in the kernel.

- If you logged in as root to test, log out and back in again as user. If you have not already done so, commit your changes to the MP1 sources.
- Type `cp /workdir/mp1/mp1.S /workdir/source/linux-2.6.22.5/drivers/char` to copy your `mp1.S` file to your kernel source directory.
- Type `cd ~/build` to change to the Linux build directory.
- Type `make` to build the kernel with your changes. If you have applied the `mp1.diff` file as described in the "Getting Started" section of this handout, the kernel will build and link properly.
- Follow the procedure described in MP0, "Preparing Your Environment," to install your new kernel onto the test virtual machine and run it. We suggest that you execute it under `gdb` when debugging.
- Navigate to your mp1 directory using command `cd /workdir/mp1` then type `make clean` and `make`.
- Type `su -c ./ktest` to execute the kernel test program as `root` (you will need to type `root`'s password).

Both test programs should produce the exact same behavior.

Moving Data to/from the Kernel

Virtual memory allows each user-level program to have the illusion of its own memory address space, separate from any other user-level program and also separate from the kernel. This affords each program a level of protection, such that user-level programs cannot write to memory owned by other programs, or worse, owned by the kernel. Therefore, when passing memory addresses between a user-level program and the kernel (such as in an `ioctl` system call) a translation is needed so that the kernel can correctly reference the user-level memory address being passed to it to get at the data. This translation is performed by the `mp1_copy_to_user` and `mp1_copy_from_user` functions, which are wrappers around the real Linux kernel functions `copy_to_user` and `copy_from_user` defined in `asm-i386/uaccess.h`.

The declarations for these two functions are:

```
unsigned long mp1_copy_to_user (void *to, const void *from, unsigned long n);
unsigned long mp1_copy_from_user (void *to, const void *from, unsigned long n);
```

The semantics of `mp1_copy_to_user` and `mp1_copy_from_user` are similar to those of `memcpy`, for those of you familiar with it. These functions take two pointers to memory areas, or **buffers**, `to` and `from`, and a length `n`. Each function copies n bytes from the `from` buffer to the `to` buffer. As can be inferred from their names, `mp1_copy_to_user` copies data from a kernel buffer to a user-level buffer, and `mp1_copy_from_user` copies data from a user-level buffer to the kernel. All user- to kernel- address translations are taken care of by these functions. Each of these functions returns the number of bytes that could not be copied, which should be 0. Bad user-level pointers can cause return values greater than zero. For example, if you pass a NULL pointer in as the user-level parameter to one of these functions (such as the `to` parameter in `mp1_copy_to_user`), it checks the pointer and memory area, sees that it points to an invalid buffer, and returns `n`, since it could not copy any data.

You'll need these functions in any of the core functions which take pointers to user-level structures. Each `ioctl` takes an "arg" parameter, so you will need to look at the documentation for each `ioctl` to figure out which ones are actually pointers to user-level structures.

One final important note: When copying data to a buffer in the kernel, you should not use statically-allocated global buffers. In multiprocessor systems, for example, multiple calls to your `ioctl` functions may be going on at the same time. Using a statically-allocated storage area, like a global variable, is a bad idea because the separate calls to the `ioctl` would be contending for using this same storage area. You should use either local variables on the stack or dynamically-allocated memory. Refer to the Course Notes for information on allocating local variables on the stack. The section below has information on dynamic memory allocation in the Linux kernel.

Allocating and Freeing Memory

User-level C programs make use of the `malloc()` and `free()` C library functions to allocate memory needed for storing dynamic structures such as linked list elements. Linux kernel code uses a number of different memory allocation functions that you will learn later in the semester. Since your code must run in the kernel, you must use the memory allocation services provided there. To abstract the details away (for now), the MP1 distribution contains two memory allocation functions that behave similarly to `malloc()` and `free()`. Their prototypes are:

```
void* mp1_malloc(unsigned long size);
void mp1_free(void* ptr);
```

`mp1_malloc` takes a parameter specifying the number of bytes of memory to allocate. It returns a `void*`, called a "void pointer," which is the memory address of the newly-allocated memory.

`mp1_free` takes a pointer to a block of memory that was allocated with `mp1_malloc()` and releases that memory back to the system. It does not return anything.

Text-Mode Video

Each character on the text display comprises two bytes in memory. The low byte contains the ASCII value for the character to be display. The high byte is an attribute byte, which holds information about the color of that particular character on the screen.

The screen is divided into rows and columns, with the upper-left character position referred to as row 0, column 0. Each row is 80 characters wide, and there are 25 rows. The screen is stored linearly in video memory, with each successive row stored directly after the one above it. For example, row 1, column 0 immediately follows row 0, column 79 in memory, row 2, column 0 immediately follows row 1, column 79, and so forth. Thus, to write a pixel at row 12, column 15 on the screen, you first need to calculate the row offset: row 12 $\times$ 80 characters per row $\times$ 2 bytes per character = 1920. Then, add the column offset: column 15 $\times$ 2 bytes per character = 30. So, row 12 column 15 on the screen is 1920 + 30 = 1950 bytes from the start of video memory.

mp1_poke: Due to Linux's virtualization of the screen buffer and of video memory, the start of video memory is not a constant, so writing to video memory is somewhat more complicated than using a `mov` instruction. To simplify things for this MP, a function has been defined called `mp1_poke`. This function, defined in assembly in `mp1.S`, takes care of finding the starting address of video memory and writing a single byte to an offset from that starting address. `mp1_poke` does not make use of the C calling convention discussed in the Course Notes. Instead, to use `mp1_poke`, make a function call with the following parameters:

`%eax`	offset from the start of video memory
`%cl`	ASCII code of character to write

`mp1_poke` then finds the correct starting address in memory and writes the character in CL to the location specified by EAX.

Note: For `mp1_poke`, EDX is a caller-saved register (*i.e.*, `mp1_poke` clobbers EDX). If you need to preserve the value of EDX across a call to `mp1_poke`, you must save its value on the stack. This preservation can be accomplished by pushing the register's value onto the stack with `pushl %edx` before making the call, and then popping the value back into EDX with `popl %edx`. All other registers are callee-saved (*i.e.*, `mp1_poke` preserves their values).

Jump Tables

You must use a **jump table** to perform the "dispatching" operation in `mp1_ioctl`. A jump table is a table in memory containing the addresses of functions (called function pointers). Each function pointer is a 32-bit memory address, just like any other pointer. The memory addresses that you want to put in the jump table are the labels of the start of each function. Let's say you have three functions in an assembly (`.S`) file, with labels `function0`, `function1`, and `function2` as the names of each. You can define a jump table as follows:

```
function0:
# function 0 body

function1:
# function 1 body

function2:
# function 2 body

jump_table:
.long function0, function1, function2
```

The jump table provides an easy way to access those three functions. If you view the jump table as a C-style array of pointers, e.g.:

```
void* jump_table[3];
```

`jump_table[0]` (in other words, the memory location at `jump_table` + 0 bytes) holds the address of `function0`, `jump_table[1]` (at `jump_table` + 4 bytes) holds the address of `function1`, and `jump_table[2]` (at `jump_table` + 8 bytes) holds the address of `function2`. In these examples, the number inside the brackets is the "index" into the jump table.

In this MP, the `cmd` parameter should serve as the index into the jump table, and you should be able to easily jump to each of the five core functions by creating a table similar to that shown above.

Linux Device Driver Overview

The first important concept in Linux device drivers is the fact that Linux makes all devices look like regular disk files. If you list the files in the `/dev` directory (using `ls`), you can see some devices that may be present on the machine. Each device is associated with one of the files. For example, the first serial port is associated with the device file `/dev/ttyS0`. For this MP, you will be dealing with the `/dev/rtc` device file, which is the device file associated with the real-time clock.

Since everything looks like a file, Linux drivers must support a certain set of standard file operations on their associated device files. These operations should seem familiar, as they are the operations available for normal disk files: `open`, `close`, `read`, `write`, `lseek` (to move to arbitrary locations within the file), and `poll` (to determine if data is available for reading or writing). In addition, most device files support the `ioctl` operation. `ioctl` is short for "I/O control," and this operation is used to perform miscellaneous control and status actions that do not easily fall into one of the more standard file operations—things that you wouldn't do to normal disk files. A good example of an `ioctl` is setting the frequency or rate of interrupts generated by the real-time clock. `ioctl`s are discussed in more depth later in this handout. It is also important to note that drivers need not support all these operations; they may choose to support only those necessary to make the device useful.

RTC Overview

A computer's real-time clock can generate interrupts at a settable frequency. Real programs running on Linux can make use of this device to perform timing-critical functionality. For example, a Tetris-style video game may need to update the positions of the falling blocks every 500 milliseconds (ms). Using the RTC, the game might set up the RTC to generate interrupts every 500 ms. Using the standard file operations above, the game can then know exactly when 500 ms has elapsed, and update its internal state accordingly.

We now use the RTC driver to illustrate how the standard file operations given above map to a real device. The RTC driver uses the `open` and `close` operations as initialization and cleanup mechanisms for certain internal data structures and setup routines. Once `open`'ed, four bytes of data become available to be read from `/dev/rtc` on every RTC interrupt. Programs can use the `read` or `poll` file operations to wait for these four bytes of data to become available, thus effectively waiting for the next RTC interrupt to be generated. The `ioctl` operation handles many other functions: setting the interrupt rate, turning RTC interrupts on and off, setting alarms, *etc.*

The important concept to glean from this discussion is that drivers provide a uniform file-like interface to the outside world via their device file and the standard set of file operations described above. The internals of actually managing the device itself are left to the driver, and are not visible to normal programs. For example, in the RTC driver, no program is able to directly gain control of the RTC, manage its interrupts, *etc.* Changing the frequency is accomplished via an `ioctl`, and determining when an interrupt has been generated is done by waiting for the four bytes of data to become available to be read using `read` or `poll`.

Ioctl Functions

An `ioctl` call from a user-mode program looks like the following:

```
ioctl(int file_descriptor, int IOCTL_COMMAND, unsigned long data_argument);
```

The `file_descriptor` parameter is returned from a call to `open` on a particular file, in this case `/dev/rtc`. It is simply a number used by a program to reference a particular file that the program has opened. The program then passes this file descriptor to other functions like `ioctl`, indicating that it is `/dev/rtc` that the program wishes to operate upon.

The `IOCTL_COMMAND` parameter is the particular `ioctl` operation to be performed on the device. It is shown in caps because all `ioctl` operations are defined as constants in the header file for each device driver. All that is needed for a program to do is select the proper predefined `ioctl` command and pass that command to the `ioctl` call.

Finally, the `data_argument` parameter is an arbitrary value passed to the `ioctl`. It can be a numeric value or a pointer to a more complex structure used by the `ioctl`. The MP1 testing code passes pointers to special structures that contain all the data necessary for your RTC driver to perform the new `ioctl`s described below.

Tasklets

Interrupt handlers themselves should be as short as possible to allow the operating system to perform other time-critical tasks. Remember, when an interrupt occurs, control is immediately handed to the operating system so it can service the device. All other tasks are blocked while the interrupt handler is executing. A tasklet is a way for an interrupt handler to defer work until after the kernel has finished processing time-critical tasks and is about to return to a user program. Normally, the interrupt handler does urgent work with the device, and then schedules a tasklet to run later to do the heavier I/O or computation that takes much longer. The operating system can enable all interrupts while the tasklet is executing. The main reason for deferring this sort of work is to allow other interrupts to occur while this non-critical work is being done. This improves the responsiveness of the system.

In MP1, the RTC hardware interrupt handler schedules your tasklet (`mp1_rtc_tasklet`) to run. When the kernel is about to return from the interrupt, it calls your tasklet, which then can update the text-mode video screen, yet allow other interrupts to occur.

Coding Style and Design

In general, being able to write readable code is a skill that's just as important as being able to write working code. People and industry teams have their own preferences and rules when it comes to coding style. In this class, we won't nitpick over small things such as spaces, newlines, camel case, *etc.*or enforce any rigorous coding guidelines. However, we still do have a basic standard that we expect you to adhere to and will be enforced through grading. Our expectations are outlined below:

- Give meaningful and descriptive (but not too long) names to your variables, labels, constants, functions, and files. It's a good idea to be consistent in your naming conventions.
- Do NOT use magic numbers (any number that appears in your code without a comment or meaningful symbolic name). -1, 0, and 1 are usually OK when used obviously.
- Keep programs and functions relatively short. Don't write spaghetti code that jumps back and forth everywhere. Create helper functions instead and make it easy to follow the flow of the program.
- Use comments to explain all functions or subroutines (i.e. function interfaces), lengthy segments of code, and any non-obvious line of code. However, do NOT overdo it. Too many comments is just as bad as too little. Use comments to explain why, not what.

Handin

Handin will consist of a demonstration in the 391 Lab. During the demo, a TA will check the functionality of your MP, review your code, and ask some basic questions to test your understanding of the code. You may also be asked to describe the RTC, tasklets, or any other concept introduced in this project.

Important Things to Note:

- Regardless of your assigned demo day, the deadline is the same for everyone!
- Once the deadline hits, your GitLab write access to the project will be revoked and you will not be able to push to your repositories.
- You are free to develop your own system of code organization, but we will **STRICTLY** use only the `master` branch for grading. All files to be graded must be in the root directory as initialized for you.