CS 1550 – Project 1: Graphics Library
Due: Sunday, January 31, 2016 by 11:59pm

Project Description
Your instructor was learning ARM assembly language one summer break and was trying to make some projects for CS/COE 0447 to do. While text-based programs are okay, it's really the graphical programs that make it a bit more interesting. So your instructor started to write a graphics library so the students could make a simple game.

In doing so, he noticed that a lot of the basic functionality the library provided was accessing the low-level operating system features that we talk about in 1550. The light bulb went off over his head, and project 1 was born.

In this project, you'll be writing a small graphics library that can set a pixel to a particular color, draw some basic shapes, and read keypresses.

How it Will Work
You will provide the following library functions (explained further below):

Library Call	System Call(s) used
void init_graphics()	open, ioctl, mmap
void exit_graphics()	ioctl
void clear_screen()	write
char getkey()	select, read
void sleep_ms(long ms)	nanosleep
void draw_pixel(int x, int y, color_t color)	
void draw_rect(int x1, int y1, int width, int height, color_t c)	
void fill_rect(int x1, int y1, int width, int height, color_t c)	
void draw_text(int x, int y, const char *text, color_t c)	

Each library function must be implemented using only the Linux syscalls. You may **not** use any C standard library functions in your library anywhere.

init_graphics()

In this function, you should do any work necessary to initialize the graphics library. This will be at least these four things:

1. You will need to open the graphics device. The memory that backs a video display is often called a **framebuffer**. In the version of Linux we are using, the kernel has been built to allow you direct access to the framebuffer. As we learned in 449, hardware devices in Unix/Linux are usually exposed to user programs via the /dev/ filesystem. In our case, we will find a file, `/dev/fb0`, that represents the first (zero-th) framebuffer attached to the computer.

   Since it appears to be a file, it can be opened using the open() syscall. To set a pixel, we only need to do basic file operations. You could seek to a location and then write some data. However, we will have a better way.

2. We can do something special to make writing to the screen easier than using a whole bunch of seeks and writes. Since each of those is a system call on their own, using them to do a lot of graphics operations would be slow. Instead, we can ask the OS to map a file into our address space so that we can treat it as an array and use loads and stores to manipulate the file contents. We will be covering this more later when we discuss memory management, but for now, we want to use this idea of **memory mapping** for our library. The mmap() system call takes a file descriptor from open (and a bunch of other parameters) and returns a void *, an address in our address space that now represents the contents of the file. That means that we can use pointer arithmetic or array subscripting to set each individual pixel, provided we know how many pixels there are and how many bytes of data is associated with each pixel.

   One note is that we must use the MAP_SHARED parameter to `mmap()` because other parts of the OS want to use the framebuffer too.

3. In order to use the memory mapping we have just established correctly, we need some information about the screen. We need to know the resolution (number of pixels) in the x and y directions as well as the bit-depth: the amount of colors we can use. The machine that QEMU is configured to emulate is in a basic 640x480 mode with 16-bit color. This means that there are 640 pixels in the x direction (0 is leftmost and 639 is rightmost) and 480 pixels in the y direction (0 is topmost, 479 is bottommost). In 16-bit color, we store the intensity of the three primary colors of additive light, Red, Green, and Blue, all packed together into a single 16-bit number. Since 16 isn't evenly divisible by three (RGB), we devote the upper 5 bits to storing red intensity (0-31), the middle 6 bits to store green intensity (0-63), and the low order 5 bits to store blue intensity (0-31).

   You can then use `typedef` to make a color type, `color_t`, that is an unsigned 16-bit value. You can also make a macro or function to encode a `color_t` from three RGB values using bit shifting and masking to make a single 16-bit number.

   To get the screen size and bits per pixels, we can use a special system call: `ioctl`. This system call is used to query and set parameters for almost any device connected to the system. You pass it a file descriptor and a particular number that represents the request you're making of that device. We will use two requests: FBIOGET_VSCREENINFO and FBIOGET_FSCREENINFO. The first will give back a `struct fb_var_screeninfo` that will give the virtual resolution. The second will give back a `struct fb_fix_screeninfo` from which we can determine the bit depth. The total size of the `mmap()`'ed file would be the `yres_virtual` field of the first struct multiplied by the `line_length` field of the second.

4. Our final step is to use the `ioctl` system call to disable keypress echo (displaying the keys as you're typing on the screen automatically) and buffering the keypresses. The commands we will need are TCGETS and TCSETS. These will yield or take a `struct termios` that describes the current terminal settings. You will want to disable canonical mode by unsetting the ICANON bit and disabling ECHO by forcing that bit to zero as well.

There is one problem with this. When the terminal settings are changed, the changes last even after the program terminates. That's why we have the exit_graphics() call. It can clean up the terminal settings by restoring them to what they were prior to us changing them.

That handles the normal termination, but it's always possible that our program abnormally terminates, i.e., it crashes. In that case when you try to type something at the commandline, it will not show up on the screen. What you're typing is still working, but you cannot see it. A reboot will fix this, but I found it useful to make a little helper program I called "fix" to turn echo back on in this event.

exit_graphics()

This is your function to undo whatever it is that needs to be cleaned up before the program exits. Many things will be cleaned up automatically if we forget, for instance files will be closed and memory can be unmapped. It's always a good idea to do it yourself with `close()` and `munmap()` though.

What you'll definitely need to do is to make an ioctl() to reenable key press echoing and buffering as described above.

clear_screen()

We will use an **ANSI escape code** to clear the screen rather than blanking it by drawing a giant rectangle of black pixels. ANSI escape codes are a sequence of characters that are not meant to be displayed as text but rather interpreted as commands to the terminal. We can print the string "\033[2J" to tell the terminal to clear itself.

getkey()

To make games, we probably want some sort of user input. We will use key press input and we can read a single character using the `read()` system call. However, `read()` is blocking and will cause our program to not draw unless the user has typed something. Instead, we want to know if there is a keypress at all, and if so, read it.

This can be done using the Linux non-blocking system call `select()`.

sleep_ms()

We will use the system call `nanosleep()` to make our program sleep between frames of graphics being drawn. From its name you can guess that it has nanosecond precision, but we don't need that level of granularity. We will instead sleep for a specified number of milliseconds and just multiply that by 1,000,000.

We do not need to worry about the call being interrupted and so the second parameter to `nanosleep()` can be NULL.

draw_pixel()

This is the main drawing code, where the work is actually done. We want to set the pixel at coordinate (x, y) to the specified color. You will use the given coordinates to scale the base address of the memory-mapped framebuffer using pointer arithmetic. The frame buffer will be stored in **row-major order**, meaning that the first row starts at offset 0 and then that is followed by the second row of pixels, and so on.

draw_rect()

Using draw_pixel, make a rectangle with corners (x1, y1), (x1+width,y1), (x1+width,y1+height), (x1, y1+height) outlined with the specified color.

fill_rect()

Make a rectangle with corners (x1, y1), (x1+width,y1), (x1+width,y1+height), (x1, y1+height) filled with the specified color.

draw_text()

Draw the string with the specified color at the starting location (x,y) – this is the upper left corner of the first letter.

We have provided (actually, it's from Apple) a font encoded into an array. It is in a header file iso_font.h that you should #include into your code. (See the section on copying files in and out of qemu below for obtaining it.)

Each letter is 8x16 pixels. It is encoded as 16 1-byte integers:

0	0	0	0	0	0	0	0	= 0x00
0	0	0	0	0	0	0	0	= 0x00
0	0	1	1	1	1	1	0	= 0x3e
0	1	1	0	0	0	1	1	= 0x63
0	1	1	0	0	0	1	1	= 0x63
0	1	1	0	0	0	1	1	= 0x63
0	1	1	1	1	1	1	1	= 0x7f
0	1	1	0	0	0	1	1	= 0x63
0	1	1	0	0	0	1	1	= 0x63
0	1	1	0	0	0	1	1	= 0x63
0	1	1	0	0	0	1	1	= 0x63
0	0	0	0	0	0	0	0	= 0x00
0	0	0	0	0	0	0	0	= 0x00
0	0	0	0	0	0	0	0	= 0x00
0	0	0	0	0	0	0	0	= 0x00

The array `iso_font` defined as a global in the `iso_font.h` file is indexed by the ASCII character value times the number of rows, so the 16 values for the letter 'A' (ASCII 65) shown above can be found at indices (65*16 + 0) to (65*16 + 15).

Using shifting and masking, go through each bit of each of the 16 rows and draw a pixel at the appropriate coordinate if the bit is 1.

You may find it convenient to break this into two functions, one for drawing a single character, and then the required one for drawing all of the characters in a string. Do not use `strlen()` since it is a C Standard Library function. Just iterate until you find `'\0'`.

You do not need to worry about line breaking.

Installing and Running QEMU

For everyone

Download the qemu-arm.zip file from the website and extract the files into a folder.

On Windows

Double-click the start.bat file in the folder to launch QEMU.

On Mac OS X

Install QEMU through Homebrew. If you don't have Homebrew, open a terminal and type:

```
ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

Go through the install steps. When done, install qemu by typing:

```
brew install qemu
```

That will install qemu. Now you can run `start.sh` in the zipped folder to launch qemu.

On Linux

Using your appropriate package manager, install qemu-system-arm, likely part of your distro's qemu package.

Then run `start.sh` in the zipped folder to launch qemu.

Setting up the Development Environment

To keep downloads small, the disk image we have provided does not have a full development environment installed. To install gcc, gdb, and ssh/scp, run the script:

```
./devtools.sh
```

When this finishes downloading and installing, you should have the ability to use most basic Linux commands, nano for text editing, and gcc, gdb, and ssh/scp/sftp for transferring files. These commands will survive a reboot, so this only needs to be done once.

If you want to install other programs, you may use the Tiny Core Linux distribution's package installer:

```
tce
```

This lets you search for a particular package or program name and install it.

After you've finished installing the tools, you may wish to make a backup of the disk image in case something happens and you don't feel like redoing all of these steps.

Shutdown linux using the command (rebooting has been turned into poweroff by an option to QEMU):

```
sudo reboot
```

Then, make a copy of `disk.qcow2` someplace safe. If things go wrong, you can always restore back to this point by replacing the `disk.qcow2` file in this directory with the one you've backed up.

Copying Files In and Out of QEMU

With the dev tools installed you can use scp (secure copy) to transfer files in and out of QEMU.

You can download the test driver program and font header from thoth with the command:

```
scp USERNAME@thoth.cs.pitt.edu:/u/OSLab/original/square.c .

scp USERNAME@thoth.cs.pitt.edu:/u/OSLab/original/iso_font.h .
```

You can backup a file named library.c to your private folder with:

```
scp library.c USERNAME@thoth.cs.pitt.edu:private
```

File Backups

One of the major contributions the university provides for the AFS filesystem is nightly backups.

Backup all the files you change under QEMU to your ~/private/ directory frequently!

Loss of work not backed up is not grounds for an extension. YOU HAVE BEEN WARNED.

Requirements and Submission

You need to submit:

- Your library.c file containing the implementation of the above functions using pure Linux system calls and no C Standard Library Calls
- An additional driver program of your design that shows the required function calls working

Make a tar.gz file named `USERNAME-project1.tar.gz`

Copy it to `~jrmst106/submit/1550/` by the deadline for credit.