Assignment Description

Your bootloader needs to load a dummy OS kernel, which will be provided for you. The assignment files include `boot0.S`, `boot1.S`, and `boot1main.c`. You need to complete those partially finished code. Each file has several tasks to complete. Additionally, please read through the other files to get a sense of what we provided for you.

Project Structure

Take some time to familiarize yourself with the project structure. The directory tree for the project follows this format:

- **project0**
  - **boot**
    - **boot0**
      - **boot0.S**: The first stage bootloader assembly file.
    - **boot1**
      - **boot1.S**: The second stage bootloader assembly file.
      - **boot1main.c**: The second stage bootloader C file.
      - **boot1lib.c**: Additional functions for the second stage bootloader.
      - **boot1lib.h**: Header file for boot1lib.c
      - **exec_kernel.S**: Assembly function which jumps to the kernel after it has been loaded.
  - **kern**
    - **init**
      - **entry.S**: The entry point into the kernel. Just prints a message for now.

Building An Image

For this assignment, you will be going through the manual build process to get a bootable image from your bootloader code. Feel free to automate the steps using a Makefile or script as you will need to build the image each time you update your bootloader. The steps to build this image roughly consists of the following: 1) compilation, 2) linking, 3) object copying, and 4) disk image creation.

Building boot0: Compilation

```
$ gcc -MD -fno-builtin -nostdinc -fno-stack-protector -Os -g -m32 -I. -c -o boot0.o boot0.S
```

The GNU compiler collection, gcc command, compiles the assembly code into object code. Please note that no standard C libraries are included here, which means you cannot use functions like printf().

Building boot0: Linking

```
$ ld -nostdlib -m elf_i386 -N -e start -Ttext 0x7c00 -o boot0.elf boot0.o
```

A linker combines all of the compiled object files into a single *binary image*, which in this case is a binary in the ELF (Executable and Linkable Format) format. For boot0 we only have a single object file to link, `boot0.o`.

The BIOS loads the boot sector into memory starting at address `0x7c00`, so this is the boot sector's load address. We set the link address by passing `-Ttext 0x7C00` to the linker, so the linker will produce the correct memory addresses in the generated code.

With the -e option, we setup the start label in the assembly file to be the first code to be executed.

Full information about this format is available in the ELF specification, but you will not need to delve very deep into the details of this format in this class. Although as a whole the format is quite powerful and complex, most of the complex parts are for supporting dynamic loading of shared libraries, which are not needed in this class.

For purposes of this class, you can consider an ELF executable to be a header with loading information, followed by several *program sections*, each of which is a contiguous chunk of code or data intended to be loaded into memory at a specified address. An ELF binary starts with a fixed-length *ELF header*, followed by a variable-length *program header* listing each of the program sections to be loaded. The program sections we're interested in are:

- `.text`: The program's executable instructions.
- `.rodata`: Read-only data, such as ASCII string constants produced by the C compiler. (We will not bother setting up the hardware to prohibit writing, however.)
- `.data`: The data section holds the program's initialized data, such as global variables declared with initializers like `int x = 5;`.

When the linker computes the memory layout of a program, it reserves space for *uninitialized* global variables, such as `int x;`, in a section called `.bss` that immediately follows `.data` in memory. C requires that "uninitialized" global variables start with a value of zero. Thus there is no need to store contents for `.bss` in the ELF binary; instead, the linker records just the address and size of the `.bss` section. The loader or the program itself must arrange to zero the `.bss` section.

Building boot0: Object Copying

```
$ objcopy -S -O binary boot0.elf boot0
```

This step removes the ELF header information and places the executable text at the beginning of a raw binary file. This is so that when the BIOS loads the boot sector into address `0x7c00`, that location contains the first executable instruction of boot0.

Building boot1: Compilation

Now, starting from the `boot/boot1` folder, the following command will compile `boot1.S`

```
$ gcc -MD -fno-builtin -nostdinc -fno-stack-protector -Os -g -m32 -I. -c -o boot1.o boot1.S
```

Now run this command as well for `boot1main.c`, `boot1lib.c`, and `exec_kernel.S` (just swap out the names in the last two command line arguments). After running the compilation step on each of those four files you should have a matching object (`.o`) file for each.

Building boot1: Linking

```
$ ld -nostdlib -m elf_i386 -N -e start -Ttext 0x7e00 -o boot1.elf boot1.o boot1main.o boot1lib.o
exec_kernel.o
```

Notice that we are now setting the text argument to `0x7e00`, so that the addresses will be configured to use this offset, whereas in boot0 we used `0x7c00`. Remember that these are the physical memory locations that boot1 and boot0 will be loaded into.

Building boot1: Object Copying

```
$ objcopy -S -O binary boot1.elf boot1
```

This step converts the ELF file to raw binary.

Building kernel: Compilation

Now, starting from the `kern/init` folder, the following command will compile `entry.S`

```
$ gcc -MD -fno-builtin -nostdinc -fno-stack-protector -D_KERN_ -Ikern -Ikern/kern -I. -m32 -O0
-c -o entry.o entry.S
```

Building kernel: Linking

```
$ ld -o kernel -nostdlib -e start -m elf_i386 -Ttext=0x00100000 entry.o -b binary
```

Notice that we are now setting the text argument to `0x00100000`, which is right after the BIOS.

Building final image: Disk Creation

For this last step, use these commands in the project root directory.

```
$ dd if=/dev/zero of=project0.img bs=512 count=256

$ parted -s project0.img "mktable msdos mkpart primary 63s -1s set 1 boot on"

$ dd if=boot/boot0/boot0 of=project0.img bs=446 count=1 conv=notrunc

$ dd if=boot/boot1/boot1 of=project0.img bs=512 count=62 seek=1 conv=notrunc

$ dd if=kern/init/kernel of=project0.img bs=512 seek=63 conv=notrunc
```

Note that this step consists of five different commands described below:

1. Use dd (data duplicator) to create an empty disk image, `bs` here means block size and `count` indicates the number of blocks. So we create a `256 * 512` byte disk image (128K).
2. Partition the image. This writes the partition table into the first sector. We set the kernel's partition to be bootable, which starts at sector 63.
3. Use dd to copy boot0 into the first block (or boot block). Notice that we shorten the block size argument so that we don't overwrite the partition table. For a sanity check, make sure boot0 isn't larger than 446 bytes.
4. Use dd to copy boot1 into the blocks after the boot block. The `seek` argument is the number of blocks to offset before the copy.
5. Use dd to copy the kernel into the blocks after the bootloader.

Simulating the x86

Instead of developing the operating system on a real PC, we use a program that faithfully emulates a complete PC. The code you write for the emulator will boot on a real PC too. Using an emulator simplifies debugging; you can, for example, set breakpoints inside of the emulated x86, which is difficult to do with the silicon version of an x86.

In COP 6611 we will use the QEMU Emulator, a modern and relatively fast emulator. While QEMU's built-in monitor provides only limited debugging support, QEMU can act as a remote debugging target for the GNU debugger (GDB), which you can use in this project to step through the early boot process.

Running our bootloader:

Now you're ready to run QEMU, supplying the file `project0.img`, created above, as the contents of the emulated PC's "virtual hard disk." This hard disk image contains both our boot loader and our kernel.

```
$ qemu-system-x86_64 -smp 1 -hda project0.img -serial mon:stdio -gdb tcp::4444 -m 512 -k en-us
```

Now since we are dealing with early boot sequences, we don't have access to standard C libraries. Hence, we don't have access to nice predefined printing functions like printf. Once the kernel gets started, typically serial printing is used; however, for the bootloader the easiest and most common way to "print" is via the VGA buffer (i.e. printing to the monitor). This is why we need a linux desktop development environment for this project, so QEMU can display a graphical monitor for us.

We provided two kinds of prints for you in this assignment so you can test your code. In `boot0.S` and `boot1.S`, `putstr` make a call for the BIOS to print out a character string. Once, we are in protected mode in boot1, we use the video region of memory directly since we don't have access to BIOS functions anymore (this is in `boot1lib.c`).

Quitting QEMU:

Press `CTRL+ALT` to release the mouse, then just close the window normally.

Debugging our bootloader:

For more serious debugging we can attach GDB to our QEMU instance. To do this we need to add the -S argument to QEMU.

```
$ qemu-system-x86_64 -smp 1 -hda project0.img -serial mon:stdio -gdb tcp::4444 -m 512 -k en-us
-S
```

With the -S flag QEMU will wait for a debugger before executing. Now in a new terminal tab, run GDB (settings will be extracted from the local `.gdbinit` file).

```
$ gdb
```

GDB will allow us to set breakpoints up, and step through, instruction by instruction. Since the first code to be executed is the BIOS, we just want to skip to boot0 or boot1.