Homework 2

Academic Honesty

Aside from the narrow exception for collaboration on homework, all work submitted in this course must be your own. Cheating and plagiarism will not be tolerated. If you have any questions about a specific case, please ask me. We will be checking for this!

NYU Poly's Policy on Academic Misconduct: http://engineering.nyu.edu/academics/code-of-conduct/academic-misconduct

Homework Notes :

General Notes:

- Read the assignment carefully, including what files to include.
- Don't assume limitations unless they are explicitly stated.
- Treat provided examples as just that, not exhaustive list of cases that should work.
- **TEST** your solutions, make sure they work. It's obvious when you didn't test the code.

Prerequisites:

Before starting on this assignment, make sure you have gone through the guide to setting your system up for development (available in the Resources section in NYU classes). You will need the following for this assignment:

1. QEMU
2. gdb
3. The i386 gcc toolchain

All of these should be installed already as part of Homework 1.

You should also download the **Makefile** we have written for this assignment, which will allow you to build your bootloader by just typing `` `make guess` ``.

Note: This assignment **does not use xv6**. You should create a new directory and put your code, and the Makefile, in that directory.

Bare Metal Assembly

When the computer first boots, the BIOS initializes the hardware, and then passes control to the *bootloader*. At this point, the CPU is in what's called 16-bit real mode, meaning that it's essentially compatible with the original 8086 PC. Generally, the OS bootloader will try to get out of real mode as quickly as possible, but for now we'll write a small program to get used to assembly programming and get a taste of how things work early on in boot.

Once the BIOS has done basic initialization of hardware, it will try to read a *boot sector* off of the boot medium. This could be a floppy disk, a hard drive, or whatever else the BIOS can figure out how to access (most modern BIOSes even allow booting off of the network). The BIOS reads 512 bytes from this device, checks to make sure it ends with the two bytes `0x55 0xAA`, loads it at address `0x7C00`, and then begins execution.

If we want to write a program that executes this early in boot, we will have to write it in 16-bit assembly and then assemble it into a binary boot sector. We can do this with GNU as (also known as `gas`; it's the standard open-source assembler, and is also used by xv6). Here's an example program that just prints "Hello world" to the console and then waits forever. It uses *BIOS interrupts* to set up the right video mode and then prints out a message character by character.

```
.code16         # Use 16-bit assembly
.globl start    # This tells the linker where we want to start executing

start:
    movw $message, %si  # load the offset of our message into %si
    movb $0x00,%ah      # 0x00 - set video mode
    movb $0x03,%al      # 0x03 - 80x25 text mode
    int $0x10           # call into the BIOS

print_char:
    lodsb           # loads a single byte from (%si) into %al and increments %si
    testb %al,%al   # checks to see if the byte is 0
    jz done         # if so, jump out (jz jumps if ZF in EFLAGS is set)
    movb $0x0E,%ah  # 0x0E is the BIOS code to print the single character
    int $0x10       # call into the BIOS using a software interrupt
    jmp print_char  # go back to the start of the loop

done:
    jmp done        # loop forever

 # The .string command inserts an ASCII string with a null terminator
message:
    .string    "Hello world"

 # This pads out the rest of the boot sector and then puts
```

```
 # the magic 0x55AA that the BIOS expects at the end, making sure
 # we end up with 512 bytes in total.
 #
 # The somewhat cryptic "(. - start)" means "the current address
 # minus the start of code", i.e. the size of the code we've written
 # so far. So this will insert as many zeroes as are needed to make
 # the boot sector 510 bytes log, and

.fill 510 - (. - start), 1, 0
.byte 0x55
.byte 0xAA
```

Make sure to read the comments (anything after a `#` symbol) and fully understand what each statement does.

If you put that code into a file called `hello.s` in the same directory as the Makefile attached, you can then create a boot sector named `hello` from it by typing `make hello`, which will invoke the assembler and linker to produce the 512 byte boot sector:

```
    $ make hello
    i386-jos-elf-as hello.s -o hello.o
    i386-jos-elf-ld -N -e start -Ttext 0x7C00 hello.o -o hello.elf
    i386-jos-elf-objcopy -O binary hello.elf hello
    rm hello.o
```

You can see that it's 512 bytes using `ls`, and (if you like) look at the raw bytes that will be loaded into memory with `xxd`:

```
    $ ls -l hello
    -rwxr-xr-x@ 1 moyix  staff  512 Feb  9 12:59 hello
    $ xxd hello
    0000000: be16 7cb4 00b0 03cd 10ac 84c0 7406 b40e  ..|.........t...
    0000010: cd10 ebf5 ebfe 4865 6c6c 6f20 776f 726c  ......Hello worl
    0000020: 6400 0000 0000 0000 0000 0000 0000 0000  d...............
    0000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
    [...]
    00001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
```

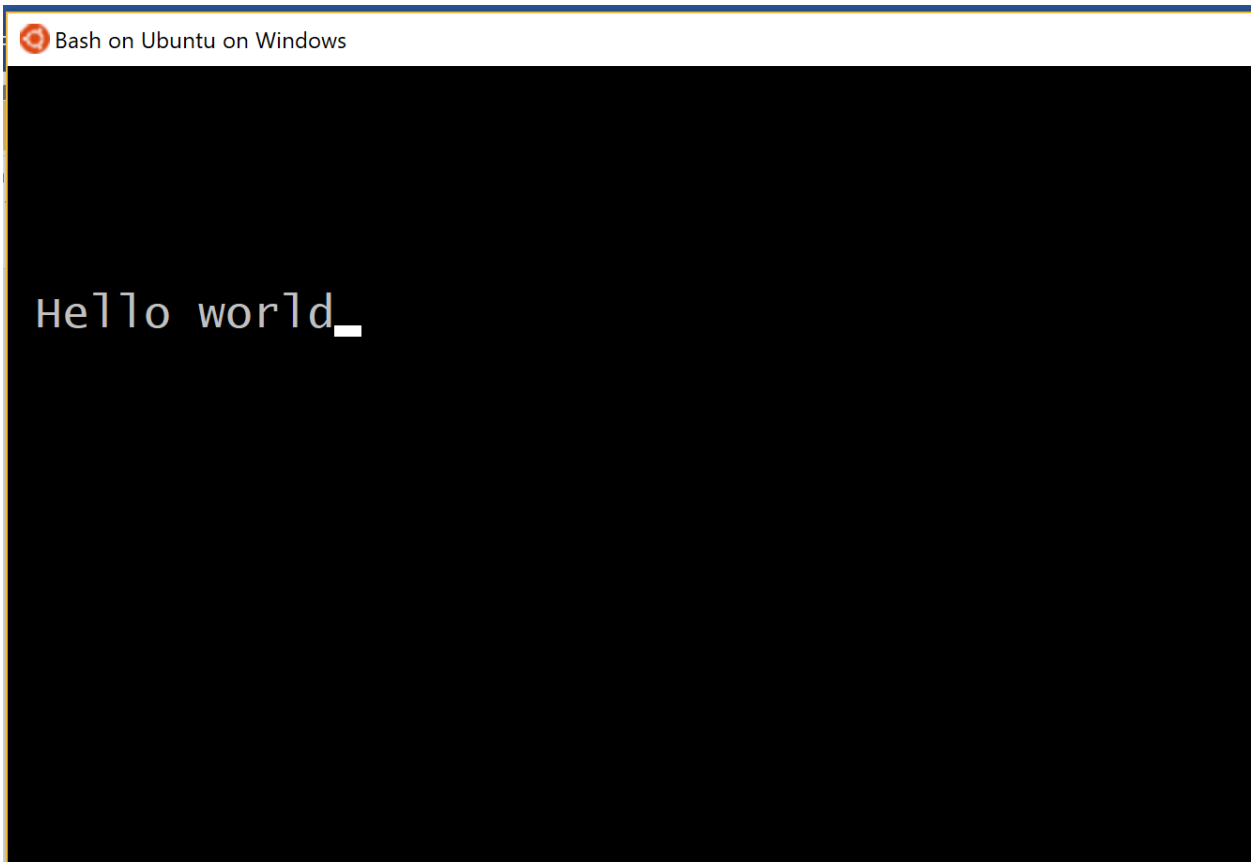
Now, run your boot sector by telling QEMU to use it as the first hard drive:

```
    Qemu-system-i386 -hda hello
```

NOTE: If you are running in Ubuntu on Windows use the following instead:

```
    Qemu-system-i386 -curses -hda hello
```

You should see a screen like this:



Homework: Guessing Game

Write an x86 boot sector that implements a simple guessing game. You should ask the user for a number from 0 to 9, read their response, and then tell them if they got it right or wrong. If they got it right, print a success message and then loop forever.

```
What number am I thinking of (0-9)? 4
Wrong!
What number am I thinking of (0-9)? 9
Wrong!
What number am I thinking of (0-9)? 2
Wrong!
What number am I thinking of (0-9)? 5
Right! Congratulations.
```

Hints:

1. Go back and look through the slides for Lecture 3 (assembly) to get a summary of the various x86 instructions. If you want more details on any of them, you can consult the Intel Architecture manual Volume 2a and Volume 2b. Remember that the instructions listed in the manual will be in Intel syntax, whereas `gas` wants AT&T syntax. The major differences are described here

2. You will need some way of generating a random number. One way to do this is to just use the seconds portion of the current time. This is accessible through a standard PC peripheral called the CMOS RTC (Real Time Clock). You can read in detail about how to interact with it here. Access to the CMOS is done through *Port I/O* (specifically, ports `0x70` and `0x71`); recall from class that this uses the x86 `in` and `out` instructions. Of course, seconds range from 0-59 and you want a number from 0-9, so you'll have to find some way to scale the result into something in the right range.

3. You will need some way of getting input from the user. We have already seen how to use a BIOS interrupt to write out a character. Another BIOS interrupt (`int 0x16`) will read a single character from the user when `AH` is set to 0 and store the ASCII value of the result in `AL`. You can read more details about this BIOS function here]. There is also a very large list of BIOS functions available here.

4. Writing a *carriage return* (ASCII 0x0d) will move the cursor to the beginning of the line. Writing a *line feed* (ASCII 0x0a) will move the cursor down one row.

5. You can make QEMU give you more information by turning on its ***debug*** output with the `-d` flag. Use `-d ?` to find out what debugging options are available. You can also put the debug log into a file with `-D debuglog.txt`.

6. You will probably want to use gdb to help debug your code. QEMU supports debugging the code it's running using gdb with the `-s` flag. You will also want to use the `-S` flag (note the capitalization) to make QEMU wait until you type `continue` in gdb before it starts executing. So run QEMU as:

```
$ qemu-system-i386 -hda guess -s -S
```

Then in another terminal window you can start gdb and attach it to QEMU:

```
$ gdb
GNU gdb (GDB) 7.8.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin14.0.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
```

```
        <http://www.gnu.org/software/gdb/documentation/>.
        For help, type "help".
        Type "apropos word" to search for commands related to "word".
        (gdb) target remote localhost:1234
        Remote debugging using localhost:1234
        0x0000fff0 in ?? ()
        (gdb) set architecture i8086
        The target architecture is assumed to be i8086
        (gdb) break *0x7c00
        Breakpoint 1 at 0x7c00
        (gdb) continue
        Breakpoint 1, 0x00007c00 in ?? ()
        (gdb) x/3i 0x7c00
        => 0x7c00:       mov    $0x0,%ah
           0x7c02:       mov    $0x3,%al
           0x7c04:       int    $0x10
```

Note the use of the `` `target remote localhost:1234` `` command to connect to QEMU, followed by `set architecture i8086` to tell gdb that you're in 16-bit mode, and finally `break *0x7c00` to make gdb stop when you reach your boot sector code.

If you need a refresher on gdb commands, you can check the cheatsheet or read a gdb tutorial. It is well worth your time to learn how to use gdb.

Submit a ***commented*** assembly file named `` `guess.s` `` that assembles to a 512-byte boot sector. The boot sector should be able to run in QEMU with the command line:

```
    $ qemu-system-i386 -hda guess
```

Further Reading

People have done some remarkable things using just the 512 bytes available in a boot sector. For example, here is a graphical demo with music in 512 bytes. If you want to try it out yourself you can download the zip file, unzip it, and then run:

```
    $ qemu-system-i386 -hda AFLAtoxin.bin -soundhw pcspk
```

(The `-soundhw pcspk` option enables QEMU's PC speaker emulation, which is necessary to get the full effect.)

If you're feeling particularly adventurous, you can try running your programs on a real PC by writing them to a USB stick and then booting from the USB stick.

Submit

Submit the assembly code you wrote (`**guess.s**`) as an attachment on NYU Classes.

Credits

This homeworks is owed to a lot of hard work by Prof. Dolan-Gavitt