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

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 an 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 setup your system through development as required in homework 1. 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.

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.
```

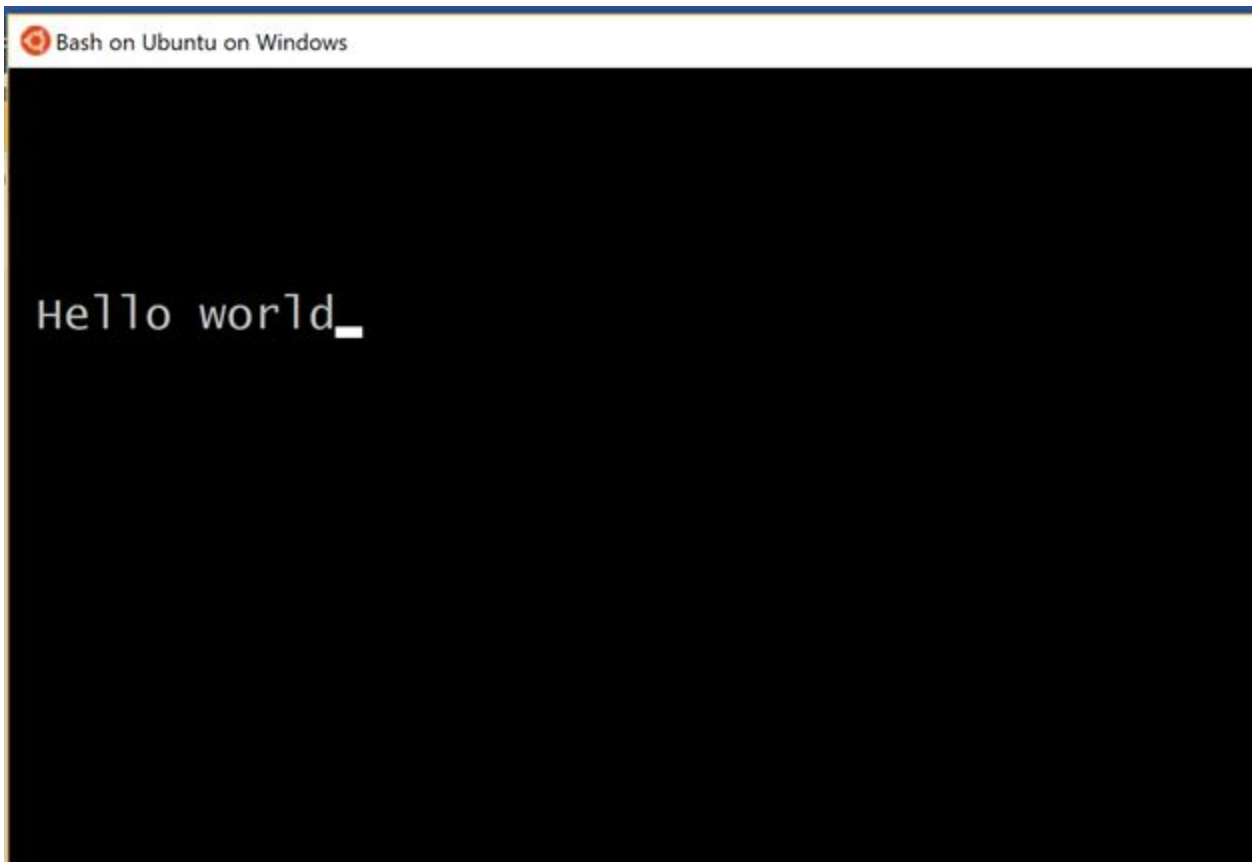
Now, run your boot sector by telling QEMU to use it as the first hard drive:

```
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:

Part 0: Setup

For this assignment, we will be trying out a new auto grader for this assignment. The way that it works, is that you will have a template repo that you will work off of on github. When you push to the repo, we will build and test your repo and email you a report to your nyu email from dev.null@os3224.nyu.cool. The reports are only so you can get feedback on whether or not your code works as intended before turning it in. Your assignment will still receive a numerical grade from a human. This will be a test run of a new system, so please bear with us if we need to work out some issues.

To fork the template repo, click on this link:

https://classroom.github.com/a/bRHoDxjk

Part 1: The Guessing Game (50 points)

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.
```

Use the provided guess.s as a template. It has all the boilerplate you will need for qemu to be able to boot. Take some time to look at the functions the makefile provides. Running make with no target will print a help screen.

Debugging with gdb:

Take advantage of gdb. There are two targets in the makefile for debugging. To start guess in debug mode, run:

```
make qemu-gdb
```

Qemu will then wait for you to connect to it. Connecting to it is as simple as running the gdb target in a new terminal window.

```
make gdb
```

Assuming all goes well, you should be dropped into a gdb shell at the beginning of your start function. For those of you not using the class VM, I would recommend that you install a gdb plugin called pwndbg. This plugin is already installed on the VM, and it will make gdb *significantly* easier to use.

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.

NOTE: The current version of pwndbg on the vm will not work with debugging guess. The fix is easy, just run the fixgdb.sh file in the repo. If you are not using the VM, just pull the most recent changes in your pwndbg directory.

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. You can use the .rand function in the guess.s template to do this. It uses the 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`.

Please note that your homework will be tested with the provided Makefile. Make sure that your homework works with the provided Makefile to avoid any silly issues.

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
```

Part 2: C Programming (30 points)

Write a program in C that when given 2 sorted linked lists of integers, merges them and returns the merged result in a new linked list. The returned Node pointer contains the Integers in ascending order.

For example:

```
$ ./linkedList 1 3 5 6 8 , 2 3 5 10 15 18
```

If lst1 = 1-->3 -->5 -->6-->8,
And lst2 = 2-->3-->5-->10-->15-->18,
calling: mergeLists (lst1, lst2), should create and return a doubly linked list that contains:
1-->2-->3-->3-->5-->5-->6-->8-->10-->15-->18.

Use the template provided (linkedList.c) and fill in the mergeLists function. To create linked lists and test with the provided template, compile the .c file and pass in the 2 linked list as command line args with a "," separating the 2 lists. This c program will **NOT** be an xv6 program. We recommend that you use either your linux environment or the class VM.

To compile linkedList, simply run:

```
$ make linkedList
```

Part 3: Short Answer Questions (20 points)

Put your responses to these questions in a file called answers.txt.

1. Difference between Monolithic and Microkernel? Also, state the advantages and disadvantages of both.
2. What do we mean by "Everything is a file in UNIX"?
3. What are file descriptors and how are they assigned? Can a file opened by the user have a file descriptor value as 2? If yes, explain how. If no, why not?

Bonus Assembly Question (5 points)

```
// Note: this is x86 32 bit (not 64 bit)
movl $0x41000000, %eax
movl $0x420000, %ebx
movl $0x4300, %ecx
xor %ebx, %eax
xor %ecx, %eax

// 1. What is the integer value in %eax?

pushl %eax              // move value of eax onto stack
movl $0x04, %eax   // syscall number for write into eax
movl $0x01, %ebx   // file descriptor for stdout into ebx
movl %esp, %ecx      // address of stack (what was in eax before)
```

```
movl $0x04, %edx   // number of bytes to write
int $0x80          // syscall

// 2. What will be printed to the screen?
```

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

Your repo is your submission. Make sure that you push to github when you are finished with the assignment. Your final push to the repo will be considered your final submission. **If you can not see your changes to the repo on github, then we can not either.**

If you submit close to the deadline, expect that you will not get an autograder report as quickly as normal.

Your repo should in the end have a completed guess.s, linkedList.c and answers.txt.

Credits

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