HOMEWORK 2

How the Operating System gets control of the machine

NOTE: This will be discussed further in class on Thursday. Reviewing some of the resources at the bottom of this page **before** the class on Thursday will help greatly.

Issued	1/21/14
Due Date	2/4/14 **before class**

The goal of this homework is to get a better understanding of what happens when a computer starts and how the operating system itself is loaded and activated. Your task is to write a floppy disk bootsector (in 8086 assembly code) which will be placed on a 1.44MB virtual floppy and attached to your virtual machine from homework 1.

Your bootsector needs to perform a number of tasks:

1. Properly initialize screen, by outputting your name, indicating that the bootsector was loaded and executed.
2. Show information about the system: how much low memory is available. What is the current time according to the machine.
3. Perform 'chain loading'. Load the bootsector from the hard drive (which was installed there earlier when installing ubuntu) and execute it. This should result in loading Ubuntu as normal.

Optional:

- Show the current time on the screen (and update!)

Some more detail

- To create an empty floppy disk image, use the following command

linux	head -c 1474560 /dev/zero > myimage.vfd
windows	fsutil file createnew myimage.vfd 1474560

Evaluation

The homework will be evaluated using the following factors:

- Correctness of assembler syntax -- The code should compile without error using **nasm**.
- Documentation of the code -- The code is clearly documented, explaining what the intent is and how it is accomplished.
- Correctness of execution -- The code performs the functions requested.
- Creativity -- Your solution is different. Make sure to **document** your code or we might miss your special trick!
- Optional tasks -- You implemented some or all of the optional requests, or went beyond what was requested in some other way.

You will be required to check in your assembly code in subversion. The state of the repository at the beginning of class on the due date will be used for evaluation.

What to hand in

As for all the other homework assignments, you will need to commit your homework to your assigned subversion repository, under the correct directory (`homework/homework2`).

Please commit the following files:

- The assembly source file (or files) (`.asm`)
- The floppy disk image (`.vfd`)
- Makefile used to build your floppy disk image
- A text file `DESCRIPTION` containing a description of what your code does, how you achieved it, clearly listing which tasks (see above) you completed.

How to create the bootsector

The end product of your efforts should be a floppy image, capable of booting an operating system installed on a harddrive. So, you don't actually need to do your development work **inside** the VM you created in homework 1. However, you will use the VM created earlier to **test** your homework.

You only need 3 things to create this floppy image:

1. An editor to edit your source code (assembly).
2. A compiler which will compile the assembly source into binary code. We will use the 'nasm' assembler.
3. A way of putting your compiled binary code in the correct location in your virtual floppy image. (Your virtual floppy image is a flat file, so the first 512 bytes of the image file will be the first 512 bytes of the 'virtual' floppy as well).

Any operating system that can perform these steps can be used to complete this homework. Nasm can be installed on Windows, Linux & OSX, so if you want, you can do all three steps on your host machine (i.e. not in the VM). However, using linux might simplify things a bit since it tends to have better development tools and more (pre-installed) tools to help automate steps 2 and 3 (such as make).

Of course, we will need to use a virtual machine to evaluate and test the bootsector. For this, you can use VirtualBox or other virtualisation software. Make sure to test your final bootsector on VirtualBox, as we will only test using VirtualBox.

If your host OS is linux, then you can do steps 1-3 using your host OS and VirtualBox to test the end result. However, if your host OS is different, you can actually use VirtualBox to provide you with the development environment, and use a second virtualbox instance to evaluate the bootsector. Note that there is a 'clone' option in VirtualBox which you can use to create the second virtualbox instance. There is no need to reinstall Ubuntu a second time.

An alternative VM: Qemu

One advantage of VirtualBox is that it is easy to use: there is a GUI which simplifies working with the virtual machines. However, this is also a disadvantage: it is harder to automate (for example, automatically booting a virtual machine to test your boot sector). VirtualBox also does not support debugging the guest.

For these reasons, another option is to use qemu (http://www.qemu.org). Qemu supports both virtualization and emulation (i.e. emulating a different architecture, including CPU). Qemu also supports connecting a debugger, allowing the **guest** (running inside qemu) to be debugged.

Qemu is freely available and can be easily installed on Ubuntu. (Use apt-get to install)

Also install gdb (if not already installed).

Using qemu, we can easily test the homework:

- We can specify the floppy image to use on the command line.
- Qemu understands the VirtualBox disk image (so we can use this as well)
- We can connect gdb to the guest running in Qemu, and debug our bootsector.

Note: the command you need to invoke is called 'qemu-system-i386' (since that is the architecture we want to emulate). Using --help will show a list of command line options. There is also a manpage (man qemu).

The '-S' and '-s' options re particularly useful for debugging: these tell qemu to expect an external debugger, and make qemu wait until a debugger attaches and instructs it to start execution.

The following gdb commands will be useful (please lookup more information in the gdb manual or google):

- target remote :1234 (tells gdb what we will be debugging)
- set history save on (helps gdb remember our earlier commands)
- set disassembly-flavor intel (tell gdb to show us nasm compatible assembler)
- set arch i8086 (tell gdb that the architecture being debugged is 8086)
- display/4i $pc (make gdb show next few instructions)
- nexti and stepi
- break *0x7c00 (make the guest stop at the beginning of the boot sector)
- continue
- info registers

Resources

The following links contain useful information:

- Good overview of the 8086 architecture
- 8086 Manual
- Description of BIOS interrupt services

- NASM documentation

Hints and advice

- Pay attention to the number representation (in assembler and documentation): hexadecimal, octal, decimal?
- nasm needs to know what mode the CPU will be in. Look for the 'BITS' directive. Likewise, a bootsector is self-contained (does not depend on external libraries). Pick the proper output format for nasm.
- BIOS interrupt 0x12, function ax=0x00 can be useful in determining how much memory is available.
- Do not alter memory under 0x500. This region is reserved for the interrupt table and bios data.
- The BIOS loads your bootsector at address 0x7c00 and executes it. For the rest, you're on your own. This means you are responsible for setting correct values for the segment registers and setting up a stack.
- Make sure that nasm and you (through segment registers) agree on what the origin of your code is (see 'org' directive)
- For chain loading: since your bootsector code (currently executing) is located where the new bootsector needs to go (0x7c00), you **might** have to move your code out of the way...
- Don't assume register values are preserved when invoking a service interrupt.
- The 'ndisasm' command can be very helpful to verify the final layout and code of your bootsector.
- Using a makefile can simplify compiling and testing your code.
- Make sure to take backups; Even better would be to often check in your code using subversion.
- In order to output a numeric value (such as a register value) to the screen, you will need to convert it to string representation. The 'div' instruction (http://asm.inightmare.org/opcodelst/index.php?op=DIV) should be very useful for this.
- The BIOS recognizes a valid bootsector by a signature (0xAA55) located at the end of the bootsector (offset 510 & 511). The following code is helpful to put the signature there:

```
; Convention: boot sector must have 0xAA55 in last 2 bytes
; See http://www.nasm.us/doc/nasmdoc3.html#section-3.5 for an
; explanation of $ and $$

   times 510-($-$$) db 0
   dw 0xAA55
```

Questions and Answers

- How are we supposed to compile our assembly code?

  Use the nasm command. (See links above). You can download nasm from the page linked in the resource section. Even easier, under ubuntu, you can use the 'apt-get' command to install the nasm package if it is not already installed.

  However, keep in mind that the output of the compilation is not designed to run under an OS. It is intended to be executed by the BIOS. This means you will not be able to simply execute it as you would with typical C program.