German University in Cairo
Faculty of Media Engineering and Technology
Dr. Aysha Alsafty
Eng. Nourhan Ehab
Eng. Eslam Osama

CSEN 602-Operating Systems, Spring 2017
Course Project - Milestone 0
Due on Friday 10/2/2017 by 11:59 pm

Project Objective

In this project, you are expected to build a tiny yet functional operating system similar to CP/M [1] from scratch. The operating system that you will implement is intended to fit on a $3\frac{1}{2}$ inch floppy disk, and to be bootable on any normal x86 PC. The operating system is to be programmed primarily in C with a small amount of assembly functions. It contains no externally written functions or libraries; all the code is written exclusively for this project.

Booting

When a computer is turned on, it goes through a process known as **booting**. The computer starts executing the Basic Input/Output System (BIOS) which comes with the computer and is stored in ROM. The BIOS loads and executes a very small program called the **bootloader**, which is located at the beginning of the disk. The bootloader then loads and executes the **kernel** a larger program that comprises the bulk of the operating system. In this project, you will write a very small kernel that will print out "Hello World" to the screen and hang up. This is a warmup milestone intended to get you familiar with the tools and simulator that you will use in the subsequent milestones.

Tools

You will need a Linux machine to complete this project (you can refer to the announcements section on the course webpage on the MET website for installation guidelines). Additionally, you need to obtain the following utilities:

- **Bochs x86 Processor Simulator:** type `sudo apt-get install bochs` and `sudo apt-get install bochs-x` in a terminal to download Bochs.
- **bcc (Bruce Evan's C Compiler):** a 16-bit C Compiler. Type `sudo apt-get install bcc` to obtain bcc.
- **as86, ld86:** A 16 bit assembler and linker that typically comes with bcc. To get as86 and ld86 type `sudo apt-get install bin86`.
- **gcc:** the standard 32-bit GNU C compiler. This generally comes with Unix.
- **nasm:** the netwide assembler. To get nasm type `sudo apt-get install nasm`.

[1] http://en.wikipedia.org/wiki/CP/M

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Aysha Alsafty
Eng. Nourhan Ehab
Eng. Eslam Osama

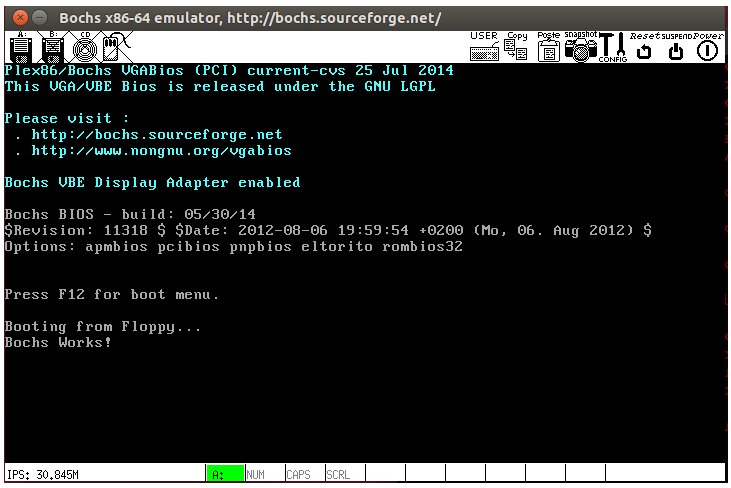
- **hexedit:** a utility that allows you to edit a file in hexadecimal byte-by-byte. To get hexedit type `sudo apt-get install hexedit`.
- **dd:** a standard low-level copying utility. This generally comes with Unix.
- A text editor.

Bochs

Bochs is a simulator of an x86 computer. It allows you to simulate an operating system without potentially wrecking a real computer in the process. To run Bochs you will need a Bochs configuration file. The configuration file tells Bochs things like what drives, peripherals, video, and memory the simulated computer is expected to have. A configuration file `conf.bxrc` is available inside `M0.zip`[2]. To run Bochs type `bochs -f config.bxrc`.

The second thing you will need is a disk image. A disk image is a single file containing all the bytes stored on a simulated floppy disk. In this project you are writing an operating system that will run off of a $3\frac{1}{2}$ inch floppy disk. You will be using a disk image to simulate the floppy disk. To get Bochs to recognize the disk, you should name the file `floppya.img` and store it in the same directory as the configuration file `config.bxrc`.

To test Bochs, rename `test.img` to `floppya.img`. You will find both files inside `M0.zip`. Make sure that `floppya.img` is in the same directory as the configuration file `config.bxrc` and run Bochs. If you see the message "Bochs works!" in the Bochs window, you are ready to proceed. Note that when you first run Bochs, the Bochs window might be completely black. If this happens, go back to the terminal and type 'c'. You should then see the following screen:



[2]Available on the MET website

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Aysha Alsafty
Eng. Nourhan Ehab
Eng. Eslam Osama

The Bootloader

The first thing that the computer does after powering on is read the bootloader from the first sector of the floppy disk into memory and start it running. A floppy disk is divided into sectors, where each sector is 512 bytes. All reading and writing to the disk must be in whole sectors; it is impossible to read or write a single byte. The bootloader is required to fit into Sector 0 of the disk, be exactly 512 bytes in size, and end with the special hexadecimal code `55 AA`. Since there is not much that can be done with a 510 byte program, the whole purpose of the bootloader is to load the larger operating system from the disk to memory and start it running.

Since bootloaders have to be very small and handle such operations as setting up registers, it does not make sense to write it in any language other than assembly. You are not required to write a bootloader in this project; one is supplied to you in `M0.zip` (`bootload.asm`). You will need to assemble it, however, and start it running.

If you look at `bootload.asm`, you will notice that it is a very small program that does three things. First it sets up the segment registers and the stack to memory `10000 hex`. This is where it puts the kernel in memory. Second, it reads 10 sectors (5120 bytes) from the disk starting at sector 3 and puts them at `10000 hex`. This would be fairly complicated if it had to talk to the disk driver directly, but fortunately the BIOS already has a disk read function prewritten. This disk read function is accessed by putting the various parameters into various registers, and calling Interrupt `0x13`. After the interrupt, the program at sectors 3-12 is now in memory at `0x10000`. The last thing that the bootloader does is it jumps to `0x10000`, starting whatever program it just placed there. That program should be the one that you are going to write. Notice that after the jump it fills out the remaining bytes with 0, and then sets the last two bytes to `55 AA`, telling the computer that this is a valid bootloader.

To install the bootloader, you first have to assemble it. The bootloader is written in `x86` assembly language understandable by the NASM assembler. To assemble it, type `nasm bootload.asm`. The output file `bootload` is the actual machine language file understandable by the computer.

You can look at the file with the `hexedit` utility. Type `hexedit bootload`. You will see a few lines of numbers, which is the machine code in hexadecimal. Below you will see a lot of 00s. At the end, you will see the magic number `55 AA`.

Next you should make an image file of a floppy disk that is filled with zeros. You can do this using the `dd` utility.
Type `dd if=/dev/zero of=floppya.img bs=512 count=2880`. This will copy 2880 blocks of 512 bytes each from `/dev/zero` and put it in file `floppya.img`. 2880 is the number of sectors on a 3 1/2 inch floppy, and `/dev/zero` is a phony file containing only zeros.

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Aysha Alsafty
Eng. Nourhan Ehab
Eng. Eslam Osama

What you will end up with is a 1.47 megabyte file `floppya.img` filled with zeros.

Finally you should copy `bootload` to the beginning of `floppya.img`. Type `dd if=bootload of=floppya.img bs=512 count=1 conv=notrunc`. If you look at `floppya.img` now with `hexedit`, you will notice that the contents of `bootload` are at the beginning of `floppya.img`.

If you want, you can try running `floppya.img` with Bochs. Nothing meaningful will happen, however, because the bootloader just loads and runs garbage. You need to write your program now and put it at sector 3 of `floppya.img`.

The Hello World Kernel

Your program in this project should be very simple. It should simply print out "Hello World" to the top left corner of the screen and stop running. You should write your program in C and call it `kernel.c`. When writing C programs for your operating system, you should note that all the C library functions, such as `printf, scanf, putchar`...etc are unavailable to you. This is because these functions make use of services provided by Linux. Since Linux will not be running when your operating system is running, these functions will not work (or even compile). You can only use the basic C commands. Stopping running is the simple part. After printing "Hello World!", you don't want anything else to run. The simplest way to tie up the computer is to put your program into an infinite while loop.

Printing hello is a little more difficult since you cannot use the C `printf` or `putchar` commands. Alternatively, the BIOS provides a software interrupt that will take care of printing **a single character** to the screen for you. Interrupt `0x10` calls the BIOS to perform a variety of I/O functions. If you call interrupt `0x10` with `0xE` in the `AH` register, the ASCII character in the `AL` register is printed to the screen at the current cursor location.

Since interrupts may only be called in assembly language, you are provided in `kernel.asm` with an assembly function `interrupt` that makes an interrupt happen. The interrupt function takes five parameters: the interrupt number, and the interrupt parameters passed in the `AX`, `BX`, `CX`, and `DX` registers, respectively[3].

To use interrupt `0x10` to print out the letter 'H', you will need to do the following:

1. Figure out the parameters of the interrupt. To print out 'H', `AH` must equal `0xE` and `AL` must equal `0x48` (the ASCII hexadecimal representation of 'H').

2. Calculate the value of `AX`. `AX` is always `AH*256 + AL`.

[3]You can find the register parameters for the various BIOS interrupts online. A good resource is `http://www.ctyme.com/intr/int.htm`

3. Call the interrupt routine. Since registers BX, CX, and DX are not used, pass 0 for those parameters. The call should look like this:
   `interrupt(0x10, 0xE*256+'H', 0, 0, 0);`

Your task is to use interrupt 0x10 to print out to the screen "Hello World!".

Compiling kernel.c

To compile your C program you cannot use the standard Linux C compiler `gcc`. This is because `gcc` generates 32-bit machine code, while the computer on start up runs only 16-bit machine code (most real operating systems force the processor to make a transition from 16-bit mode to 32-bit mode, but we are not going to do this). `bcc` is a 16-bit C compiler. `bcc` is fairly primitive and requires you to use the early Kernighan and Ritchie C syntax rather than later dialects. You should not expect programs that compile with `gcc` to necessarily compile with `bcc`.

To compile your kernel, type `bcc -ansi -c -o kernel.o kernel.c`. The `-c` flag tells the compiler not to use any preexisting C libraries. The `-o` flag tells it to produce an output file called `kernel.o`.

`kernel.o` is not your final machine code file, however. It needs to be linked with `kernel.asm` so that the final file contains both your code and the `int10()` assembly function. You will need to type two more lines:

```
as86 kernel.asm -o kernel_asm.o
ld86 -o kernel -d kernel.o kernel_asm.o
```

The first line assembles `kernel.asm`, while the second line links `kernel.o` and `kernel_asm.o` and produce `kernel`. The file `kernel` is your program in machine code. To run it, you will need to copy it to `floppya.img` at sector 3, where the bootloader is expecting to load it (in later projects you will find out why sector 3 and not sector 1).

To copy it, type `dd if=kernel of=floppya.img bs=512 conv=notrunc seek=3` where "seek=3" tells it to copy kernel to the third sector. Try running Bochs. If your program is correct, you should see "Hello World" printed out.

Scripts

Notice that producing a final `floppya.img` file requires you to type several lines that are very tedious to type over and over. An easier alternative is to make a Linux shell script file. A shell script is simply a sequence of commands that can be executed by typing one name. To generate a script, put all the previous commands into a single text file, with one command per line, and call it `compileOS.sh`. Then type `chmod +x compileOS.sh`, which tells Linux that `compileOS.sh` is an executable. Now, when you change `kernel.c` and want to recompile, simply type `./compileOS.sh`.

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Aysha Alsafty
Eng. Nourhan Ehab
Eng. Eslam Osama

Project Teams

You should work in teams of **exactly four**. You can form teams from different tutorials. You will work in the same team for all the milestones of the project. Kindly submit your team members names and IDs in the following form `https://goo.gl/forms/o1zKhSjOCR8Y6My93` by latest **Friday 3/2/2017 11:59 pm**. Note that if you form teams of less than four members, other members will be assigned randomly. Anyone who does not form a team before the mentioned deadline will be assigned randomly to a team and will be forced to work with them till the end of the semester. The final teams list will be published on the MET website on Saturday 4/2. No changes will be implemented afterwards.

Project Deliverables and Submission

For this milestone you are required to submit ONE zip containing all of your files. You should use this webform `https://podio.com/webforms/17713857/1190666` to submit your project. The project should be submitted as ONE zip folder containing both files. Late submissions will not be accepted.

Have fun :)