**German University in Cairo**
**Faculty of Media Engineering and Technology**
**Dr. Amr Desouky**
**Eng. Nourhan Ehab**
**Eng. Rana Helal**

# **CSEN 602-Operating Systems**, Spring 2016
**Course Project - Milestone 1**
Due on Thursday 19/2/2015 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 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.

## Milestone 1: Booting
When a computer is turned on, it goes through a process known as **booting**. The computer starts executing the 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. 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

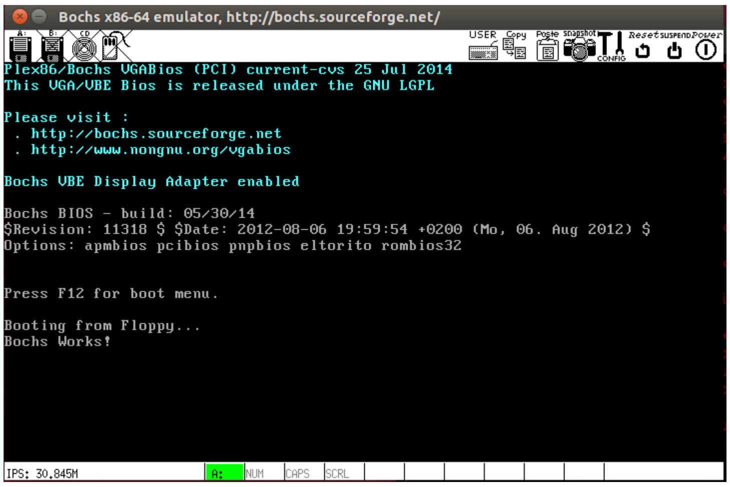
- **hexedit:** a utility that allows you to edit a file in hexadecimal byte-by-byte. To get hexedit type `sudo apt-get install hexedit`.
- **dd:** a standard low-level copying utility. This generally comes with Unix.
- A text editor.

## 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 `M1.zip`[2]. To run Bochs type `bochs -f conf.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 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.

To test Bochs, rename `test.img` to `floppya.img`. You will find both files inside `M1.zip`. Make sure that `floppya.img` is in the same directory as the configuration file `bochs.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:



```
Plex86/Bochs VGABios (PCI) current-cvs 25 Jul 2014
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 05/30/14
$Revision: 11318 $ $Date: 2012-08-06 19:59:54 +0200 (Mo, 06. Aug 2012) $
Options: apmbios pcibios pnpbios eltorito rombios32


Press F12 for boot menu.

Booting from Floppy...
Bochs Works!
```

[2]Available on the MET website

**German University in Cairo**
**Faculty of Media Engineering and Technology**
**Dr. Amr Desouky**
**Eng. Nourhan Ehab**
**Eng. Rana Helal**

## 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 `M1.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.

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, 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. Instead, you have to write directly to video memory. Video memory starts at address `0xB8000`. Every byte of video memory refers to the location of a character on the screen. In text mode, the screen is organized as 25 lines with 80 characters per line. Each character takes up two bytes of video memory: the first byte is the ASCII code for the character, and the second byte tells what color to draw the character. The memory is organized line-by-line. Thus, to draw the letter 'A' at the beginning of the third line down, you would have to do the following:

- Compute the address relative to the beginning of video memory:
  80 * (3-1) = 160.
- Multiply that by 2 bytes/character: 160 * 2 = 320.
- Convert that to hexadecimal: 320 = 0x140
- Add that to B8000: 0xB8000+0x140 = 0xB8140 - this is the address in memory
- Write the letter 'A' to address 0xB8140. The letter A is represented in ASCII by the hexadecimal number 0x41.
- Write the color white (0x7) to address 0xB8141.

**German University in Cairo**
**Faculty of Media Engineering and Technology**
**Dr. Amr Desouky**
**Eng. Nourhan Ehab**
**Eng. Rana Helal**

Since 16 bit C provides no built in mechanism for writing to memory, you are provided with an assembly file `kernel.asm`. This file contains a function `putInMemory`, which writes a byte to memory. The function `putInMemory` can be called from your C program and takes three parameters:

1. The first hexadecimal digit of the address, times `0x1000`. This is called the `segment`.
2. The remaining four hexadecimal digits of the address.
3. The character to be written

Accordingly, to print out the letter A in white at the beginning of the third line of the screen, you should call:

```
putInMemory(0xB000, 0x8140, 'A');
putInMemory(0xB000, 0x8141, 0x7);
```

Your task now will be to write a C kernel program to print out "Hello World" at the **top left corner** of the screen.

**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`.

## 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 `http://goo.gl/forms/qcVX9GUgu0` by latest **Thursday 11/2/2015 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.

## Project Deliverables and Submission

For this milestone you are required to submit `kernel.c` and `floppya.img`.. You should use this webform `https://goo.gl/i04vAP` to submit your project. The project should be submitted as ONE zip folder containing both files. Late submissions will not be accepted.

Have fun :)