Operating Systems

Project #1: Introduction & Booting

Objective

This project will familiarize you with the boot process and also with the tools and the simulator that you will use in subsequent projects. The end product for this project will be (1) a very small kernel that will print out "Hello World" to the screen and hang up (i.e., enter an infinite loop) and (2) a shell script for creating & running a bootable disk image containing that kernel.

Background

When a computer is turned on, it goes through a process known as booting. The computer starts executing a small program, known as the bootstrap program, which is contained in the BIOS (**b**asic **i**nput/**o**utput **s**ystem), which comes with the computer and is stored in ROM. The BIOS bootstrap program reads the first sector of the boot disk and loads it into memory. The first sector of the boot disk will contain a small program called the *bootloader*. After loading the bootloader into memory, the BIOS performs a jump to the address where it placed the bootloader, which starts the bootloader program executing. When executed, the bootloader loads and executes the operating system *kernel*--a larger program that comprises the bulk of the operating system.

Tools

You will use the following tools to complete this and subsequent projects:

- **bochs** - An x86 processor emulator (pronounced box)
- **bcc** (Bruce Evan's C Compiler) - A 16-bit C compiler
- **as86** and **ld86** - A 16-bit 0x86 assembler and linker
- **gcc**/**g++** - The standard 32-bit GNU C compiler
- **nasm** - The Netwide Assembler
- **hexedit**/**hexdump** - Utilities that allows you to edit/view a file in hexadecimal byte-by-byte
- **dd** - A standard low-level copying utility.
- A simple text-based editor of your choice (e.g., gedit, vim, xemacs, sublime)

All of these tools are available for free for Mac and Linux (but not for Windows). They may already be installed (or part of the project). If not, you are welcome to download and install them on your own Mac/Linux machines. However, I cannot provide any significant support for the download and installation of the tools on your personal machines.

Getting Started

We have created a VM image with everything set up and ready to go. VirtualBox is recommended for Macs, and VMware Player is recommended for Windows. Follow the directions, but use the VM for OS instead. If the 64-bit one doesn't work for you, e-mail me to get the 32-bit version.

Once you start the VM and open a terminal, you will find the following files in the `os/project1` folder:

- `bootload.asm` – assembly code for the boot loader.
- `kernel.asm` – assembly language routines you will use in your kernel.
- `opsys.bxrc` – bochs configuration file.
- `test.img` – bootable 1.44MB floppy image for testing bochs.

Booting bochs

Open a terminal window (or use the one from above):

1. Change into your `project1` directory
2. Copy `test.img` into `floppya.img`
3. Type the command: `bochs -f opsys.bxrc`

This command will run bochs using the `opsys.bxrc` configuration file. The configuration file tells bochs things like what drives, peripherals, video, and memory the simulated computer has. It also tells bochs where to boot from, in this case from the file `floppya.img`. The `floppya.img` file is a 1.4MB floppy disk image, which is a file that contains exactly the

number of bytes that can be stored on a 1.4MB floppy disk. This particular image contains the assembled boot loader and a small kernel that simply prints out “Bochs works!”.

After executing the `bochs -f opsys.bxrc` command, you will need to hit return, or 6 and return (on UNIX, square braces are used to indicate the default `[option]`) to begin the simulation. If all goes according to plan, when bochs boots from `floppya.img` the bootloader will load and execute this small kernel you will see the message “Bochs works!” appear in the bochs window. If you do not see the expected results repeat the above steps, double-checking your work.

The Bootloader

We will be booting bochs from a 1.44MB floppy disk image. A 1.4MB floppy disk has 2880 sectors of 512 bytes. Thus, the bootloader is required to be exactly 512 bytes long (one sector) and be loaded into sector 0 of the boot disk image. In addition, the last two bytes of sector 0 must be `0x55` followed by `0xAA`, which indicates to the BIOS that the disk is a *boot disk*. Since not much can be done with a 510 byte program, the purpose of the bootloader is to load the larger operating system from the disk to memory and start it running.

Since a bootloader has 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. Consequently you are not required to write a bootloader in this project; one is supplied to you in the file `bootload.asm`. You will however need to assemble it and install it into sector 0 of your boot disk image.

I encourage you to open the `bootload.asm` file in a text editor and study its contents. It is a very small program that does three things:

1. The bootloader sets up the segment registers and the stack to memory address `0x10000`. This is where it puts the kernel in memory
2. It reads 10 sectors (5120 bytes) from the disk starting at sector 3 and puts them at address `0x10000`. 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 parameters into various registers, and calling Interrupt 13 (hex). After the interrupt, the program at sectors 3-12 is now in memory at address `0x10000`.
3. It jumps to `0x10000`, starting whatever program it just placed there. In this project that program will be a small kernel that prints “Hello World.” In future projects it will be your actual OS kernel. Notice that after the jump it fills out the remaining bytes with 0, and then sets the last two bytes to `0x55` followed by `0xAA`, telling the computer that this is a valid bootloader. NOTE: the assembly code contains the number `0xAA55`. The Intel architecture stores data in *little endian* form, i.e., the least significant bytes are stored in at lower addresses, which means the hex digits will be stored from right to left as we go from lower addresses to higher addresses.

Assembling the 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, use the command:

```
nasm bootload.asm
```

The nasm assembler generates the output file `bootload`, which contains the actual machine language program that is understandable by the computer. You can look at the `bootload` file with the `hexdump` utility. You will see a few lines of numbers, which are the machine language instructions in hexadecimal. Below that you will see a lot of 00s. Near the end, you will see the magic number `55 AA` indicating that it is a boot sector. (Alternatively, you could run `hexdump` with the -x option, which will display in two-byte hexadecimal.)

Disk Images

We will use the unix `dd` utility to create disk images. The first thing we'll do is create a disk image filled with all 0's. To do this, use the command:

```
dd if=/dev/zero of=floppya.img bs=512 count=2880
```

The above command will copy `count=2880` sectors of `bs=512` bytes/sector from the input file `if=/dev/zero` and put it in the output file `of=floppya.img`. 2880 is the number of sectors on a 1.4MB 3.5" floppy, and `/dev/zero` is a unix special file that contains only zeros. You should end up with a 1.4 MB file named `floppya.img` that is filled with zeros.

We will also use the `dd` utility to copy the `bootload` program to sector 0 of the `floppya.img` disk image. To do this use the command:

```
dd if=bootload of=floppya.img bs=512 count=1 conv=notrunc seek=0
```

The additional parameters to `dd` indicate that we move `seek=0` sectors before writing and that we do not truncate the image after writing (`conv=notrunc`). If you look at `floppya.img` now with `hexdump`, the contents of bootload are contained in the first 512 bytes of `floppya.img` and the rest of the file is filled with 0's.

If you want, you can try booting bochs using `floppya.img`. However, nothing meaningful will happen, because the bootloader in sector 0 will just load sectors 3-10, which contain all 0's, and then attempt to run them.

In the next part of the assignment you'll write your "Hello World" program and put it into sector 3 of `floppya.img` so that it runs when bochs is booted.

A Hello World Kernel

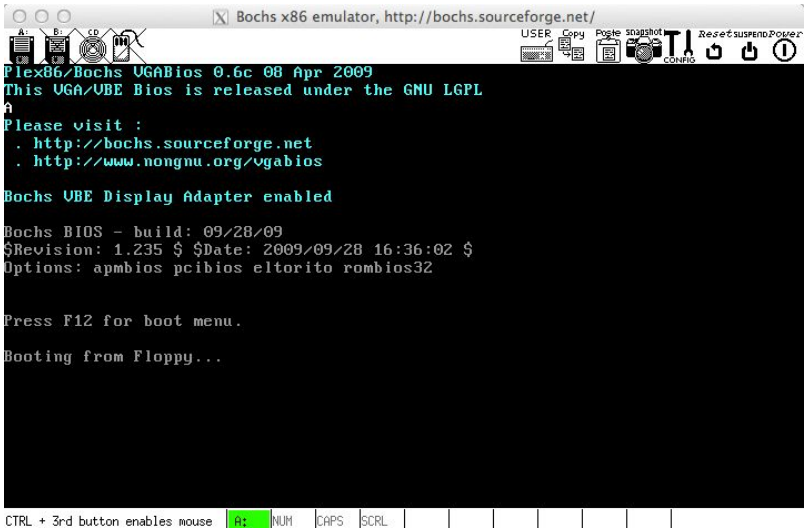
For this project, the kernel should contain a `main` method that simply prints out "Hello World" in white letters on a black background at the top left corner of the screen and then enter an infinite while loop. You must write your program in C and save its source code in a file named `kernel.c`.

When writing C programs for an existing operating system such as Mac OS, Linux or Windows, you can use functions such as `printf` or `putchar` that display text on the screen (similar to `System.out.println` in Java). When these functions are compiled, they use a system call to the operating system, which ultimately handles the display of the characters. But, since we don't have an OS yet (you haven't written it!), we can't have system calls. Thus there is no way to compile calls to functions such as `printf` and `putchar` that rely on system calls for their operation.

Because **you cannot use the `printf` or `putchar` functions,** you will have to write the characters you want to display directly to video memory. Video memory starts at memory 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 use to draw the character. The memory is organized line-by-line.

Thus, to draw a white letter 'A' on a black background at the beginning of the third line down, you would have to do the following:

1. Compute the address relative to the beginning of video memory:
   - Since one line is 80 characters long, the beginning of the third line down would be 80*(3-1) = 160
2. Multiply that relative location by **2 bytes / character**: 160*2 = 320
3. Convert that to hexadecimal (e.g., using a converter: http://www.binaryhexconverter.com/decimal-to-hex-converter): 320 = `0x140`
   - Note: your implementation will not literally have to do this.
4. Add that to the starting address of video memory (0xB8000) to get the memory address: `0xB8000+0x140 = 0xB8140`
5. Write `0x41`, the ASCII code for the letter 'A', to address `0xB8140`.
   - ASCII: 65, binary: 0100 0001, hex: 0x41
6. Write the color white (`0x0F`) to address `0xB8141`



See the A up in the corner?

The 16-bit C compiler that we are using provides no built-in mechanism for writing bytes directly to memory. To allow you to write bytes to memory from your kernel, you are provided with an assembly file kernel.asm that contains the function `putInMemory`. This function has the following signature:

```
void putInMemory(int segment, int offset, char b)
```

- `segment` - the most significant hex digit of the address times `0x1000`.
- `offset` - The four least significant hex digits of the address.
- `b` - the ASCII code of the character to be written.

To write the character 'A' to address `0xB8140`, you could call:

```
putInMemory(0xB000, 0x8140, 65);
```

Fortunately, you will not need to translate characters to ASCII manually. In C a character is equivalent to its ASCII value. Thus, the above line can be written as:

```
putInMemory(0xB000, 0x8140, 'A')
```

Alternatively, I can let C do the hex conversion for me, e.g.,

```
putInMemory(0xB000, 0x8000 + 320, 'A')
```

You should now be able to write a kernel that prints out “Hello World” at the top left corner of the screen before entering an infinite while loop.

Compiling the Kernel

The bochs simulator, as well as a physical PC machine, starts up in 16-bit mode. This

means that our kernel must be a 16-bit program, as opposed to a 32-bit or 64-bit program. The implication of writing a 16-bit program is that we cannot use the standard GNU `gcc` C compiler because it generates 32- or 64-bit machine language. Instead we will use the `bcc` compiler that generates 16-bit machine language code. Unfortunately, `bcc` is fairly primitive and requires that we use early C syntax. The most significant aspect of which is that all local variables used in a function must be defined before any statements in the method.

As an aside, modern 32-bit or 64-bit operating systems get around the fact that machines boot in 16-bit mode by using a secondary bootloader. The secondary bootloader is a 16-bit program that loads the 32- or 64-bit kernel into memory and then switches the machine into 32 or 64-bit mode before executing the kernel. Our OS will be a 16-bit OS so we will not have to worry about switching modes.

To compile your kernel use the command:

```
bcc -ansi -c -o kernel.o kernel.c
```

The `-c` flag tells the compiler not to use functions from any pre-existing C libraries that might rely on system calls (e.g. `printf` or `putchar`). The `-ansi` flag tells it to use standard ANSI C syntax and the `-o` flag tells it to produce an output file called `kernel.o`.

The `kernel.o` file is not your final machine code file, however. Recall, that you are using the `putInMemory` function from the `kernel.asm` file. For your C program to be able to call `putInMemory`, you will also need to assemble the `kernel.asm` file and then link it with your `kernel.o` file.

To assemble the `kernel.asm` file use the command:

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

To link the `kernel.o` and `kernel_asm.o` files into the executable kernel file use the command:

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

The file `kernel` is your program in machine code. To run it, you will need to copy it into the disk image at sector 3, where the bootloader is expecting to find it (in later projects you will find out why sector 3 and not sector 1). To copy the kernel file to sector 3, use the command:

```
dd if=kernel of=floppya.img bs=512 conv=notrunc seek=3
```

Now if you run bochs, the bootloader will load your kernel from sector 3 and execute it. If your kernel program is correct, you will see “Hello World” printed in the top left corner of

the screen.

Shell Scripts

Producing a final bootable floppy disk image requires you to type quite a few commands. Instead of typing them all each time we change the kernel, we're going to create a **bash script**. You could also use a Makefile for this step.

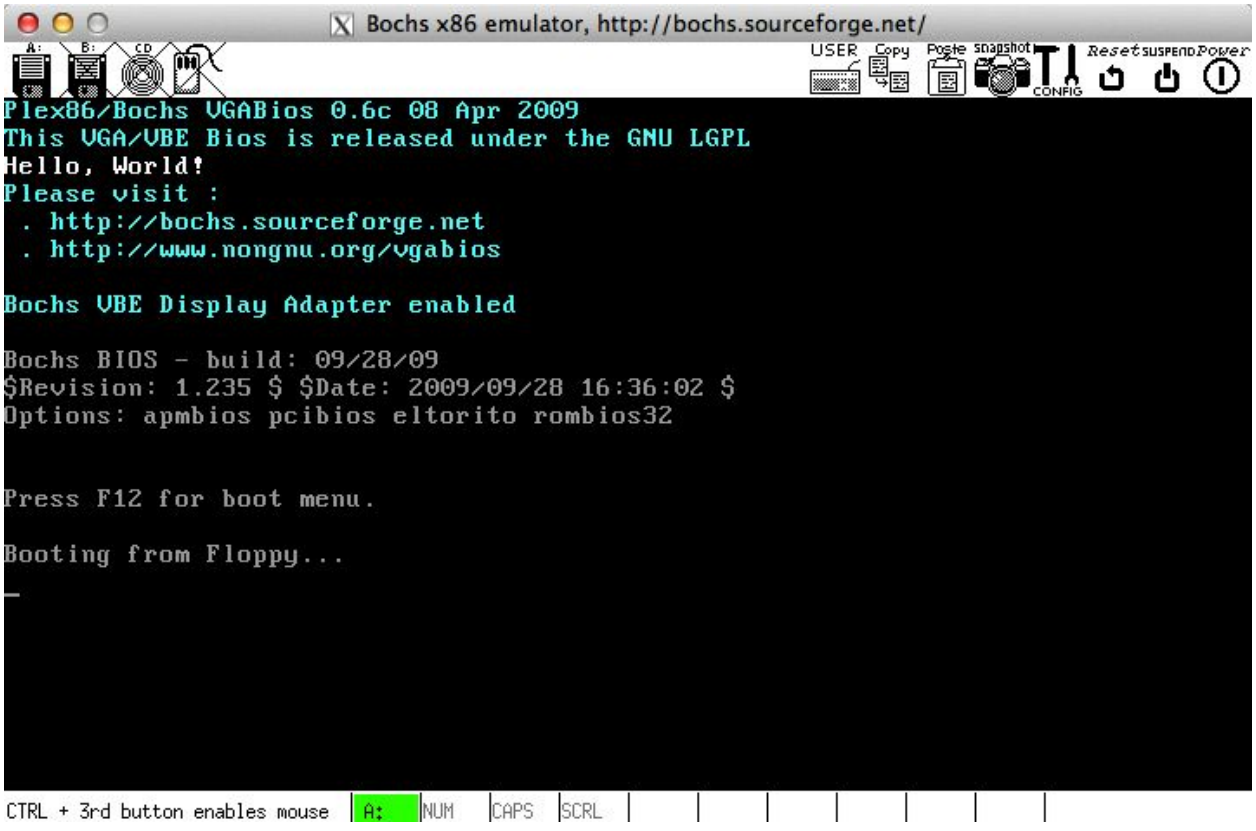
To turn the text file into a shell script, make it executable using the command:

```
chmod +x run.sh
```

Then to execute your shell script you simply type:

```
./run.sh
```

If you put the correct commands into your shell script, running the shell script will recompile your kernel, link it with the `putInMemory` function, and produce a new bootable disk image and run the simulator. ***Bonus****:* add an `if` to your script to only run the simulator if compilation with `bcc` succeeded, or create a Makefile. Even if using a Makefile, please include a simple `run.sh` to run the proper target.



Kernel Improvements

1. Create a function **`putChar`** in `kernel.c` that displays a character in a specified color at a specified location on the screen. The `putChar` function should accept as parameters

- the character to be printed
- the color in which to print it
- the row and column at which to print it.

Modify your `main` method so that, in addition to printing "Hello World" in the upper left corner of the screen, it uses the `putChar` method to display "Hello World" in white on a red background at the center of the screen. The video memory's color codes are shown below. The background color goes in the high-order nibble of the byte for the color, while the foreground color goes in the low-order nibble of the byte. For example, `0000 0001 = 0x01` will produce blue text on a black background.

0	0000	black	8	1000	dark gray
1	0001	blue	9	1001	light blue
2	0010	green	A	1010	light green
3	0011	cyan	B	1011	light cyan
4	0100	red	C	1100	light red
5	0101	magenta	D	1101	light magenta
6	0110	brown	E	1110	yellow
7	0111	light gray	F	1111	white

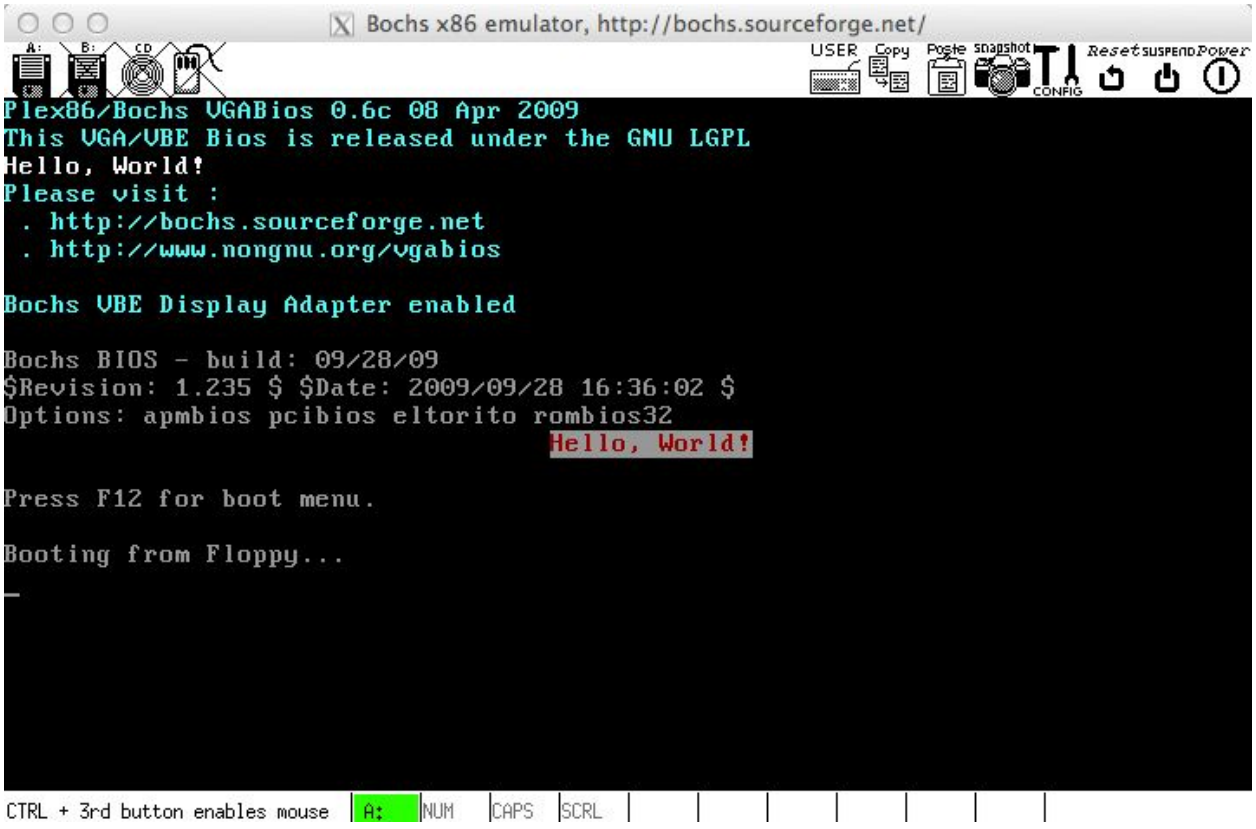
2. Create a function **`putStr`** in `kernel.c` that displays a string in a specified color at a specified location on the screen. The `putStr` function should accept as parameters

- the terminated string to be printed,
- the color in which to print it,
- the row and column at which to print it

Each successive character in the string should be printed one column to the right of the previous one. If the end of a line is reached, the next character should appear on the following line. No characters should be printed past the end of the screen. Modify your main method so that it also uses the `putStr` method to display "Hello World" in red on a white background in the lower right corner of the screen.

Tips:

- When adding functions to your kernel, add prototypes at the top of the file and then put your function definitions after main. You will see really odd behavior otherwise. Example of a prototype:
  - `void putChar(char ch, int color, int row, int column);`
- You can iterate over every character until you see the end of the string, rather than needing to get the string's length (which can be error prone).
- Don't forget to put an infinite loop at the end of main.



Bonus

Modify the `bootload.asm` assembly program so that it will allow you to dual boot. The modified bootloader should display a very short menu, something like:

```
a. HW
b. IR
```

And wait for the user to press either a or b. If the user presses `a`, the "Hello World" kernel developed in this project should be booted from sector 3 of the disk. If the user presses `b`, a different kernel that prints "I Rock" on the screen should be booted from sector 2850 of the disk. Note: You'll need to do some research to find out how to map between absolute sector numbers (like 2850) and the C:H:S addressing required for the BIOS call.

To complete this bonus, you will need to learn a bit of Intel assembler as well as a bit about the available BIOS routines for displaying characters and reading input from the keyboard. Here are a few references that might help you get started:

- A page describing a simple bootloader that displays some text on the screen using the BIOS: `http://www.digitalthreat.net/?p=303`
- A page that describes the BIOS interrupts that are available (see interrupt `0x10` with `ah=0x0E`): `http://www.ctyme.com/intr/int.htm`

Before working on this bonus, make a complete copy of your project solution. Then work on the bonus using the copy. You will be required to submit your original solution separately.

Submission

Place your `kernel.c` and `run.sh` files into a folder named `project1_name`, filling in `name` with your e-mail username. Create a zip archive of that folder and submit it to the course assignment page. If you completed the bonus, place your two kernel source files, your new dual boot `bootload.asm` file and a `run.sh` script that produces a dual bootable floppy image in another folder named bonus, zip it, and submit it as well.

Your project will be graded on its correctness as well as its style. Your `kernel.c` and `run.sh` files should be nicely formatted and well documented.

Acknowledgement

This assignment as well as the accompanying files and source code have been adopted with minor adaptations from those developed by Michael Black at American University. His paper "Build an operating system from scratch: a project for an introductory operating systems course" can be found in the ACM Digital Library at: http://dl.acm.org/citation.cfm?id=1509022.