O.S. Lab 1: Booting

We will discuss it in more detail later, but a floppy disk is divided into *sectors*, each of which 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. When an x86-based 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 *read-only memory* (*ROM*))[1]. The BIOS loads and executes a very small program called the *boot loader*, which is located in the first sector of the floppy disk. The boot loader then loads and executes the *kernel*, a larger program that comprises the bulk of the operating system, but we're getting ahead of ourselves. The purpose of this lab is to create a very small disk image and boot from it.

The Boot Loader

The boot loader 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 boot loader is to load the larger operating system from the disk to memory and start it running.

Since boot loaders 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. Consequently you are not required to write a boot loader in this project - one is supplied to you online (**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 0x10000 (hexadecimal 10000 or decimal 65536). This is where it puts the kernel in memory. Second, it reads 10 sectors (5120 bytes) from the disk starting at sector 259 and puts them at 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 various parameters into various registers, and calling Interrupt 13 (hex). After the interrupt, the program at sectors 259 through 268 is now in memory at 0x10000. (When we fully describe the file system we will explain why the kernel is stored there.) The last thing that the boot loader does is jump 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 boot loader.

To install the boot loader, you first have to assemble it. The boot loader 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.

[1] Macs don't have BIOS and some lower-end PCs use either emulated BIOS or the UEFI method. This project assumes an honest-to-goodness PC.

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 3½ 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
```

Now type **hexdump –C floppya.img**. You should see the following:

```
00000000  b8 00 10 8e d8 8e d0 8e  c0 b8 f0 ff 89 c4 89 c5  |................|
00000010  b1 04 b6 00 b5 00 b4 02  b0 0a b2 00 bb 00 00 cd  |................|
00000020  13 ea 00 00 00 10 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00168000
```

See that the contents of **bootload** are at the beginning of the **floppya.img** file.

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

Scripts Part I

Notice that producing even an empty **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 begin a script, put all commands that appear above (and below) in grey boxes into a single text file, with one command per line, and call it **compileOS.sh**. Once created, make the file executable via the command **chmod +x compileOS.sh**. We will continue to build this up as we move forward.

Interrupt-Driven Output

One of the major services an operating system provides are system calls. You will now learn how to use some of the system calls provided by the BIOS. You will then write your own system calls to print a string to the video. Once this is in place, you will write a very small kernel that will print out "Hello World" to the screen and hang up. This will create the foundation needed for the next part of the project.

Introduction to the Kernel

When writing C programs for your operating system, you should note that all the C library functions, such as **printf**, **scanf**, **putchar**... 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.

Download the additional files **kernel.c** and **lib.asm** (kernel.asm version 1) from the lab web page. Rename **lib.asm** as **kernel.asm** and open **kernel.c** in a text editor. A shortened version of this initial kernel seed file appears at right.

As you can see, stopping running is the simple part right now. After performing I/O, 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 and stop the simulator by hand. We now have to figure out what to do with the I/O.

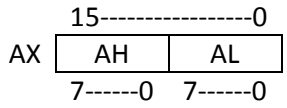
```
void printString(char*);
void printLogo();

void main()
{
  makeInterrupt21();
  printLogo();
  printString("Hello world\r\n\0");
  while (1) ;
}

/* more code follows */
```

Overview: x86 Registers

Registers are named storage locations directly inside the CPU, designed to be accessed at much higher speed than RAM. In the Intel x86 architecture the *general-purpose* or *data registers* AX, BX, CX and DX are used for arithmetic and data movement. Each of these is a 16-bit register but can be addressed as their upper or lower 8-bit values. For example AX is a 16-bit register whose upper 8 bits are called AH and whose lower 8 bits are called AL.



Thus instructions can address either the 4 16-bit registers or the 8 8-bit quantities. Changes to the 16-bit registers also modify the corresponding 8-bit registers and vice versa.

Printing to the Screen – Interrupt 16 (0x10)/Function 14 (0x0E)

The BIOS provides a software interrupt that will take care of printing to the screen for you. Interrupt 16 calls the BIOS to perform a variety of I/O functions. The one of current interest: if you call interrupt 16 with 14 (0xE) in the AH register, the ASCII character stored 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 with a 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. It returns the value returned from the interrupt in the AL register.

To use interrupt 10 to print out the letter 'Q', you will need to do the following:

```
char al = 'Q';
char ah = 14;
int ax = ah * 256 + al;
interrupt(16, ax, 0, 0, 0);
```

- Figure out the parameters.
  - To print out 'Q', AH must equal 14 and AL must equal 'Q' (81 decimal or 0x51)
  - Calculate the value of AX. AX is always AH*256 + AL.
- Call the interrupt routine. Since registers BX, CX, and DX are not used, pass 0 for those parameters.

Alternately you could simply write: **interrupt(16, 14*256+'Q', 0, 0, 0);**

Your Task

To complete step 1, you need to add a *void printString(char*,int)* function to the kernel. Your *printString* takes a C-string (i.e. character array) as a parameter (plus an extra integer that we will use later). The last character in the array should be the unprintable character '\0' (0x0). Your function should print out each character of the array until it reaches 0x0, at which point it should stop.

Notes:

- When adding functions to your C program, make sure they always follow *main()*. *main()* must always be your first function.
- When adding a function, you will need to declare it at the top. Don't forget prototypes.

Compiling the Kernel

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. It is fairly primitive and requires you to use the early Kernighan and Ritchie C syntax rather than later dialects. For example, you have to use /* */ for all comments; // is not recognized. 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 **interrupt()** assembly function. First assemble **kernel.asm** with the command **as86 kernel.asm –o kasm.o**. The type the line

```
ld86 -o kernel -d kernel.o kasm.o
```

to link all files together and produce the file **kernel**, your program in machine code. To run it, you will need to copy it to **floppya.img** at sector 259, where the boot loader is expecting to load it (in later projects you will find out why sector 259 and not sector 1). To copy it, type

```
dd if=kernel of=floppya.img bs=512 conv=notrunc seek=259
```

where **seek=259** tells it to copy kernel to the correct sector.

Try running Bochs. If your program is correct, you should see "Hello World" printed out.

Printing to the Parallel Port – Interrupt 23 (0x17)/Function 0 (0x00)

As with the screen, you can send characters to the parallel port and thus the connected printer via BIOS interrupt 23. In Bochs this is simulated with output to the file *printer.out*. For example to send the letter 'Q' to the printer simply write: **interrupt(23, 'Q', 0, 0, 0);**

Your Task

Edit the *printString()* do that, if the last parameter is 1, output goes to the printer instead of the screen. Test by printing the BlackDOS logo to screen but sending "Hello world" to the printer.

BlackDOS Function Calls

Historically MS-DOS provided many easy-to-use function calls for providing both console/keyboard and disk I/O, all supported by interrupt 33 (0x21). One advantage was that then any user program (not just the o.s. kernel) could access this functionality. In this lab we will add a couple of extra I/O services to our o.s., enable interrupt 33 and then tie everything done to date to this interrupt.

Creating Interrupt 33

Creating an interrupt service routine is simply a matter of creating a function, and putting the address of that function in the correct entry of the interrupt vector table. The interrupt vector table sits at the absolute bottom of memory and contains a 4 byte address for each interrupt number. To add a service routine for interrupt 33, write a function to be called on interrupt 33, and then put the address of that function at the right memory address.

Unfortunately, this really has to be done in assembly code. The file **kernel.asm** provides you with two functions for this. **makeInterrupt21()** simply sets up the interrupt 33 service routine. Function *interrupt21ServiceRoutine()* is henceforth automatically called whenever an interrupt 33 happens. It calls a function in your C code **void handleInterrupt21(int ax, int bx, int cx, int dx)** that you will need to remove

comments from. The AX, BX, CX, DX parameters passed in the interrupt call will show up in your *handleInterrupt21()* function as parameters ax, bx, cx, dx.

In **kernel.c** comment out the return statement and un-comment **case 0** and the **default** of the **switch** statement. Delete the *printString()* prototype. From now on the interrupt 33 handler provide all developed services. We will write the handler so that the number of the requested service is stored in AX, all arguments in BX, CX and DX. For this first instruction:

Function	AX=	BX=	CX=	Action
Print string	0	Address of string to print	Destination: 0 = screen 1 = printer	Call **printString(bx,cx)**

If AX is any other value print an error message. Now rewrite the main() function of **kernel.c** to appear like the code at right. First enable interrupt 33. Next replace all function calls with interrupt calls. It should work exactly like it did previously.

Finally edit the kernel (as at right) so that "Hello world from *your name*" goes to the printer, while the BlackDOS logo appears on screen.

```
void handleInterrupt21(int,int,int,int);
void printLogo();

void main()
{
  makeInterrupt21();
  printLogo();
  interrupt(33,0,"Hello world from your name\r\n\0",1,0);
  while (1) ;
}

/* more stuff follows */
```

Conclusion

When finished, submit a .tar or .zip file (no .rar files) containing all needed files (**bootload**, **bdos.txt**, **kasm.o**, **compileOS.sh**, **kernel.c** and **README**) to the drop box. **README** should explain what you did and how the t.a. can verify it. Your .zip/.tar file name should be your name. Be sure to include your name under "Author" in the opening banner of the kernel.

Last updated 10.29.2018 by T. O'Neil, based on material by M. Black and G. Braught. Previous revisions 1.12.2018, 12.21.2016, 2.11.2016, 1.16.2016, 1.14.2015, 2.21.2014.