CSC 565 - Operating Systems
Spring, 2008

Project A - Booting

Objective

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 warm-up project intended to get you familiar with the tools and simulator that you will use in subsequent projects.

Tools

You will need the following utilities to complete this and subsequent projects:

- Bochs 2.3.5 - An x86 processor simulator
- PuTTY or some other SSH client - essential if you don't have a Linux computer
- bcc (Bruce Evan's C Compiler) - A 16-bit C compiler
- as86, ld86 - A 16 bit assembler and linker that typically comes with bcc
- gcc - The standard 32-bit GNU C compiler
- nasm - The Netwide Assembler
- hexedit - A utility that allows you to edit a file in hexadecimal byte-by-byte
- dd - A standard low-level copying utility. This generally comes with UNIX.
- pico, nano, or vi - A simple text-based editor

All the utilities except the first two are available for Linux but not for Windows. You will need either a Linux computer or an account on a Linux machine.

If you have a Linux machine, you should obtain all of the above utilities except PuTTY. Chances are good that your operating system already has gcc, dd, and nano installed. You may already have some of

the others installed too.

If you have a Windows machine, you will need a Linux account. The CAP department server has all of the above utilities already installed. You will need to obtain an account on the department server (talk to your professor). Once you have an account, you should use an SSH utility (such as PuTTY) to connect to the server and log in to your account. You should be able to complete the project on your account.

For those of you who use Windows, you may want to install Bochs on your own computer. In my opinion, Bochs is much easier to use under Windows than it is under Linux. You will need to obtain an SFTP program (such as WinSCP) to transfer simulation files from your Linux account to your computer so that Bochs can run them.

Bochs

Bochs is a simulator of a x86 computer. It allows you to simulate an operating system without potentially wrecking a real computer in the process. It is sufficiently elaborate to run both Linux and Windows 95!

To run Bochs you will need a Bochs configuration file. The configuration file tells things like what drives, peripherals, video, and memory the simulated computer has. Configuration files for Linux and Windows are available on Blackboard. To run bochs in Linux, type "bochs -f opsys.bxrc". To run bochs in Windows, double click on opsys.bxrc.

The second thing you will need is a disk image. A disk image is a single file containing every single byte 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. 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, go to Blackboard and download test.img and opsys.bxrc. Rename test.img to floppya.img. Start bochs running. If you see the message "Bochs works!" appear in the Bochs window, you are ready to proceed.

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. Consequently you are not required to write a bootloader in this project - one is supplied to you on Blackboard (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 13 (hex). After the interrupt, the program at sectors 3-12 is now in memory at 10000. The last thing that the bootloader does is it jumps to 10000, 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.

A 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... 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 B8000 hex. 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:

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

Since 16 bit C provides no built in mechanism for writing to memory, you are provided with on Blackboard 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

To write 'A' to address 0xB8140, you should call:

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

To print out the letter A in white at the beginning of the third line of the screen, call:

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

You should now be able 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. 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 int10() assembly function. You will need to type two more lines:

as86 kernel.asm -o kernel_asm.o to assemble kernel.asm
ld86 -o kernel -d kernel.o kernel_asm.o to link them all together 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, simple type *./compileOS.sh*.

Running on a Real Computer (optional)

If your program runs correctly with Bochs, and you have a computer with a 3 1/2 inch drive, you might try running your program on a real computer.

If your computer is running Linux, you can use dd to copy. Type

dd if=floppya.img of=/dev/fd0 bs=512 count=50

Then try booting the computer off the floppy disk.

If your computer is running Windows, you will need to use *debug.exe*, located in the windows/system32 directory. First, you will need to make your floppya.img file smaller so that debug can handle it. In Linux, type *dd if=/dev/zero of=floppya.img bs=512 count=1 seek=50*. Then transfer your program to Windows. Open your program with *debug.exe* (you may have to associate .img files with debug.exe first) Then type *w 100 0 00 30*.

Submission

You should submit, by the deadline, your *kernel.c* file on Blackboard Digital Dropbox. Please include a comment at the top of the file with your name.