CPSC 422/522: Operating Systems

Homework: bootstrap and x86 assembly

This lecture covers the basics of x86 assembly language and the PC architecture, which will be important for understanding the xv6 source code and the kernel you will be building in this course. This homework will help prepare you by walking you through parts of the PC's BIOS initialization and boot process. It is somewhat longer than you can expect most of the lecture homeworks to be, but it is vital that you absorb these fundamentals thoroughly in order to understand the material in the rest of the course.

Beyond xv6 or this course, familiarity with processor architecture principles and assembly language is critical to understanding operating systems, because the OS must interact with and manage processor architecture features that are not directly supported by high-level languages (even C), such as processor configuration and privilege level switching mechanisms. Just as important is knowing the conventions by which high-level languages like C use the processor: how the C compiler implements function calls and uses its stack, how C program code and data is arranged in memory, etc.

Hand-In Procedure

You are to turn in this homework during lecture. Please write up your answers to the exercises below and hand them in to a staff member at the beginning of lecture.

Since the associated lecture is during Yale's shopping period, we will accept this homework late with no penalty up through the end of shopping period, but to aid your understanding the lecture we strongly encourage you to complete the homework on time if possible.

Assignment

First, read Kai Li's PC Architecture guide and xv6 Chapter 1. Then, if you are uncertain about your understanding of x86 assembly basics, look over and familiarize yourself with at least one of the more complete x86 architecture references listed on the reference page. Certainly the definitive reference for x86 assembly language programming is Intel's instruction set architecture reference: this course's reference page contains a somewhat older 2003 version, which is smaller and easier to navigate than the latest versions while covering all of the processor features we will use in this course. For the "latest and greatest" architecture specifications, featuring the 64-bit extensions introduced by AMD and adopted by Intel, see Intel's or AMD's web sites.

The PC's Physical Address Space

We will now dive into a bit more detail about how a PC starts up. A PC's physical address space is hard-wired to have the following general layout:

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

The first PCs, which were based on the 16-bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical address space of an early PC would therefore start at 0x00000000 but end at 0x000FFFFF instead of 0xFFFFFFFF. The 640KB area marked "Low Memory" was the *only* random-access memory (RAM) that an early PC could use; in fact the very earliest PCs only could be configured with 16KB, 32KB, or 64KB of RAM!

The 384KB area from 0x000A0000 through 0x000FFFFF was reserved by the hardware for special uses such as video display buffers and firmware held in non-volatile memory. The most important part of this reserved area is the Basic Input/Output System (BIOS), which occupies the 64KB region from 0x000F0000 through 0x000FFFFF. In early PCs the BIOS was held in true read-only memory (ROM), but current PCs store the BIOS in updateable flash memory. The BIOS is responsible for performing basic system initialization such as activating the video card and checking the amount of memory installed. After performing this initialization, the BIOS loads the operating system from some appropriate location such as floppy disk, hard disk, CD-ROM, or the network, and passes control of the machine to the operating system.

When Intel finally "broke the one megabyte barrier" with the 80286 and 80386 processors, which supported 16MB and 4GB physical address spaces respectively, the PC architects nevertheless preserved the original layout for the low 1MB of physical address space in order to ensure backward compatibility with existing software. Modern PCs therefore have a "hole" in physical memory from 0x000A0000 to 0x00100000, dividing RAM into "low" or "conventional memory" (the first 640KB) and "extended memory" (everything else). In addition, some space at the very top of the PC's 32-bit physical address space, above all physical RAM, is now commonly reserved by the BIOS for use by 32-bit PCI devices.

More recently, with the introduction of PCs that can address more than 4GB of physical RAM via 64-bit physical addresses, the PC architecture has effectively acquired a *second* hole at the top of 32-bit address space, where some 32-bit I/O devices are typically mapped, and physical RAM continues beyond this point in 64GB address space. In multiprocessor and multicore PCs, for example, this region at the top of 32-bit address space contains mappings of the Advanced Programmable Interrupt Controllers (APICs), which different processors use to signal each other and distribute I/O device interrupts among processors. See the Intel architecture manuals and the Intel MultiProcessor Specification for more information about APICs and multiprocessors. Dealing with complicated physical address spaces and other aspects of hardware organization that evolved over many years is one of the important practical challenges of OS development.

The ROM BIOS

You will now use QEMU's debugging facilities to investigate how a PC boots.

First open two terminal windows, and run `make qemu-gdb` in one and `gdb` in the other to debug xv6 remotely, as described in the previous homework assignment. This time let's look more closely at GDB's output immediately after it connects to the QEMU process running xv6:

```
$ gdb
GNU gdb (GDB; openSUSE 11.1) 6.8.50.20081120-cvs
...
+ target remote localhost:27296
0x0000fff0 in ?? ()
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
(gdb)
```

The following line is GDB's disassembly of the first instruction to be executed.

```
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
```

From this output you can conclude a few things:

- The IBM PC starts executing at physical address 0x000ffff0, which is at the very top of the 64KB area reserved for the ROM BIOS.
- The PC starts executing with `CS = 0xf000` and `IP = 0xfff0`.
- The first instruction to be executed is a `jmp` instruction, which jumps to the segmented address `CS = 0xf000` and `IP = 0xe05b`.

Why does QEMU start like this? This is how Intel designed the 8088 processor, which IBM used in their original PC. Because the BIOS in a PC is "hard-wired" to the physical address range 0x000f0000-0x000fffff, this design ensures that the BIOS always gets control of the machine first after power-up or any system restart - which is crucial because on power-up there *is* no other software anywhere in the machine's RAM that the processor could execute. The QEMU emulator comes with its own BIOS, which it places at this location in the processor's simulated physical address space. On processor reset, the (simulated) processor enters real mode and sets CS to 0xf000 and the IP to 0xfff0, so that execution begins at that (CS:IP) segment address.

How does the segmented address 0xf000:fff0 turn into a physical address? To answer that we need to know a bit about real mode addressing. In real mode (the mode that PC starts off in), address translation works according to the formula: *physical address* = 16 * *segment* + *offset*. So, when the PC sets CS to 0xf000 and IP to 0xfff0, the physical address referenced is:

```
   16 * 0xf000 + 0xfff0   # in hex multiplication by 16 is
   = 0xf0000 + 0xfff0     # easy--just append a 0.
   = 0xffff0
```

`0xffff0` is 16 bytes before the end of the BIOS (`0x100000`). Therefore we shouldn't be surprised that the first thing that the BIOS does is `jmp` backwards to an earlier location in the BIOS; after all how much could it accomplish in just 16 bytes?

Use GDB's **`si`** (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

Turn in: a disassembly of the BIOS from the target of the initial `jmp` up to the next control transfer instruction.

Since GDB has no source code line information about the BIOS, it will complain if you try to use the '`l`' (list) command, or even the simple '`disas`' (disassemble) command with zero or one operand, since it will look for a C function in the xv6 kernel corresponding to this instruction address and of course not find any. To examine "raw instructions" for which GCC has source code information (besides the immediate next instruction to be executed, which GDB prints automatically), you use the `x/i` command. This command has the syntax `x/Ni ADDR`, where *N* is the number of consecutive instructions to disassemble and *ADDR* is the memory address at which to start disassembling.

When the BIOS runs, it sets up an interrupt descriptor table and initializes various devices such as the VGA display. This is where the "`Plex86/Bochs VGABios`" messages you see in the QEMU window come from.

After initializing the PCI bus and all the important devices the BIOS knows about, it searches for a bootable device such as a floppy, hard drive, or CD-ROM. Eventually, when it finds a bootable disk, the BIOS reads the *boot loader* from the disk and transfers control to it.

The Boot Loader

Floppy and hard disks for PCs are divided into 512 byte regions called *sectors*. A sector is the disk's minimum transfer granularity: each read or write operation must be one or more sectors in size and aligned on a sector boundary. If the disk is bootable, the first sector is called the *boot sector*, since this is where the boot loader code resides. When the BIOS finds a bootable floppy or hard disk, it loads the 512-byte boot sector into memory at physical addresses 0x7c00 through 0x7dff, and then uses a `jmp` instruction to set the CS:IP to `0000:7c00`, passing control to the boot loader. Like the BIOS load address, these addresses are fairly arbitrary - but they are fixed and standardized for PCs.

The ability to boot from a CD-ROM came much later during the evolution of the PC, and as a result the PC architects took the opportunity to rethink the boot process slightly. As a result, the way a modern BIOS boots from a CD-ROM is a bit more complicated (and more powerful). CD-ROMs use a sector size of 2048 bytes instead of 512, and the BIOS can load a much larger boot image from the disk into memory (not just one sector) before transferring control to it. For more information, see the "El Torito" Bootable CD-ROM Format Specification.

For this course, however, we will use the conventional hard drive boot mechanism, which means that our boot loader must fit into a measly 512 bytes. The xv6 boot loader consists of one assembly language source file, `bootasm.S`, and one C source file, `bootmain.c` Look through these source files carefully and make sure you understand what's going on. The boot loader must perform two main functions:

1. First, the boot loader switches the processor from real mode to *32-bit protected mode*, because it is only in this mode that software can access all the memory above 1MB in the processor's physical address space. Protected mode is described briefly in sections 1.2.7 and 1.2.8 of PC Assembly Language, and in great detail in the Intel architecture manuals. At this point you only have to understand that translation of segmented addresses (segment:offset pairs) into physical addresses happens differently in protected mode, and that after the transition offsets are 32 bits instead of 16.
2. Second, the boot loader reads the kernel from the hard disk by directly accessing the IDE disk device registers via the x86's special I/O instructions. If you would like to understand better what the particular I/O instructions here mean, check out the "IDE hard drive controller" section on the reference page. You will not need to learn much about programming specific devices in this class: writing device drivers is in practice a very important part of OS development, but from a conceptual or architectural viewpoint it is also one of the least interesting.

After you understand the boot loader source code, look at the file `bootblock.asm`. This file is a disassembly of the boot loader that xv6's Makefile creates *after* compiling the boot loader. This disassembly file makes it easy to see exactly where in physical memory all of the boot loader's code resides, and makes it easier to track what's happening while stepping through the boot loader in GDB.

You can set address breakpoints at arbitrary instructions in GDB with the `b` command: for example, **`b *0x7c00`** sets a breakpoint at address 0x7C00. Once at a breakpoint, you can continue execution using the **`c`** and **`si`** commands: **`c`** causes QEMU to continue execution until the next breakpoint (or until you press **`Ctrl-C`** in GDB), and **`si`** ***`N`*** steps through the instructions *`N`* at a time.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. Trace through the execution of `bootasm.S` under GDB, referring to the source code and the disassembly file `bootblock.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Then trace into `bootmain()` in `bootmain.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

Be able to answer the following questions:

- At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
- How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

> **Turn in:** the values of the CS and EIP registers immediately before and after the processor executes the `ljmp` instruction in `bootasm.S` where the boot code jumps to a 32-bit code segment. Be sure you understand what they mean. Hint: GDB's '`info reg`' command may be useful.

Pointers in C

We will next look in further detail at the C language portion of the boot loader, in `bootmain.c`, which loads the kernel. But to understand this code thoroughly, it will be helpful to stop and review some of the basics of C programming.

Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). You may wish to purchase this book or check out a copy on reserve at the Becton Library. Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for l3-pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in output lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C available online, though not as definitive, such as a tutorial by Ted Jensen, and the Pointers section in Marshall's *Programming in C*.

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery later in this course, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

> **Turn in:** a diagram of the x86 memory layout of the \``int a[4]`' array from the l3-pointers.c program. Show the contents of the array as a sequence of *bytes* (not `int`s!), and show the exact

values of each byte in this array immediately before the statement '`c = (int *) ((char *) c + 1);`' at line 32, and immediately after the statement '`*c = 500;`' at line 33 in the program. Circle the bytes of the array that this statement changed. You may find a calculator with hex/decimal conversion capability helpful: e.g., **`gcalctool`** or **`kcalc`** on the Zoo machines.

Loading the Kernel

To make sense out of `bootmain.c` you'll need to know what an ELF binary is. When you compile and link a C program such as the xv6 kernel, the compiler transforms each C source ('`.c`') file into an *object* ('`.o`') file containing assembly language instructions encoded in the binary format expected by the hardware. The linker then combines all of the compiled object files into a single *binary image* such as `kernel`, which in this case is a binary in the ELF format, which stands for "Executable and Linkable Format".

Full information about this format is available in the ELF specification on our reference page, but you will not need to delve very deeply into the details of this format in this class. Although as a whole the format is quite powerful and complex, most of the complex parts are for supporting dynamic loading of shared libraries, which we will not do.

For purposes of this course, you can consider an ELF executable to be a header with loading information, followed by several *program sections*, each of which is a contiguous chunk of code or data intended to be loaded into memory at a specified address. The boot loader does not modify the code or data; it loads it into memory and starts executing it.

An ELF binary starts with a fixed-length *ELF header*, followed by a variable-length *program header* listing each of the program sections to be loaded. The C definitions for these ELF headers are in `inc/elf.h`. The program sections we're interested in are:

- `.text`: The program's executable instructions.
- `.rodata`: Read-only data, such as ASCII string constants produced by the C compiler. (We will not bother setting up the hardware to prohibit writing, however.)
- `.data`: The data section holds the program's initialized data, such as global variables declared with initializers like `int x = 5;`.

When the linker computes the memory layout of a program, it reserves space for *uninitialized* global variables, such as `int x;`, in a section called `.bss` that immediately follows `.data` in memory. C requires that "uninitialized" global variables start with a value of zero. Thus there is no need to store contents for `.bss` in the ELF binary; instead, the linker records just the address and size of the `.bss` section. The loader or the program itself must arrange to zero the `.bss` section.

You can display a full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:

```
$ objdump -h kernel
```

(Use `i386-elf-objdump` instead of `objdump` if you are not running on a machine like Linux that has native ELF support.)

You will see many more sections than the ones we listed above, but the others are not important for our purposes. Most of the others are to hold debugging information, which is typically included in the program's executable file but not loaded into memory by the program loader.

Besides the section information, there is one more field in the ELF header that is important to us, named `e_entry`. This field holds the link address of the *entry point* in the program: the memory address in the program's text section at which the program should begin executing. You can see the entry point:

```
$ objdump -f kernel
```

To examine memory in GDB, you use the `x` command with different arguments. The GDB manual has full details. For now, it is enough to know that the recipe `x/Nx ADDR` prints *N* words of memory at *ADDR*. (Note that both 'x's in the command are lowercase.)

Warning: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

Turn in: Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question; just think.)

Bryan Ford, Department of Computer Science, Yale University
[an error occurred while processing this directive] *Based on MIT 6.828 materials by Frans Kaashoek and others*