Lab 1

PC Bootstrap and Calling Conventions

The files for this lab (lab1.tar.gz) are located in the labs section for this course.

Introduction

This lab is split into two parts. The first part concentrates on the PC bootstrap procedure, particularly link address, load address, writing position independent code, and relocation. It should give you a good sense of how the boot loader takes the boot-time CPU state and prepares the CPU to run actual C code.

The second part checks your understanding of the GCC calling conventions and stack layout for the x86. You will write an implementation of `setjmp` and `longjmp`.

Getting Started with x86 assembly

If you are not already familiar with x86 assembly Paul A. Carter's *PC Assembly Language* is an excellent place to start. Hopefully, the book contains mixture of new and old material for you. We recommend reading the entire book, except you should skip all sections after 1.3.5 in chapter 1, and you can also skip chapters 5 and 6, and all sections under 7.2.

Warning: Unfortunately the examples in the book are written for the NASM assembler, whereas we will be using the GNU assembler. NASM uses the so-called *Intel* syntax while GNU uses the *AT&T* syntax. While semantically equivalent, an assembly file will differ quite a lot, at least superficially, depending on which syntax is used. Luckily the conversion between the two is pretty simple, and is covered in this reference. Only read the section "The Syntax".

Surely the definitive reference is The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference.

You should read the recommended chapters of the PC assembly book, and "The Syntax" section of the second reference now. Save the Intel document for later.

Part 1: PC Bootstrap

Boot Time

After install bochs,you need to modify the file lab1/boot/.bochsrc,change the roomimage and vgaromimage option into proper path.

Start bochs using our first example bootstrap block:

```
[root@oslab root]$ cd /lab1/boot
[root@oslab root]$ make ex1.disk
...
```

[root@oslab root]$ **bochs**
You should see:

```
Next at t=0
(0) f000:fff0: e968e0: jmp +#e068
<bochs:1>
```

From this you can conclude a few things:

- The IBM PC starts executing at `CS = 0xf000` and `IP = 0xfff0`.
- The first instruction executed is a relative `jmp`, which jumps by the amount +#e068. (You know this is the first instruction because "Next at t=0"). A relative jump means that this value is added to the IP. And since this value has the high-bit set, we can see that this particular `jmp` will be backwards--to a lower IP.
- The machine code for the instruction is 0xe968e0.
- The whole instruction is 3 bytes long.

Why does the Bochs start like this? This is how Intel designed the 8088 processor, which IBM used in their original PC. The PC hardware is set up so that the address range 0xf0000 - 0xfffff contains the BIOS (Basic Input/Output System), which early PCs held in ROM but current PCs store in updateable flash memory. The Bochs simulator comes with its own "virtual" BIOS, which it maps at this location in the processor's physical memory. On reset, the processor sets CS to 0xf000 and the IP to 0xfff0, and consequently, execution begins at that (CS:IP) segment address. But what memory location does this CS:IP correspond to?

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: `ADDRESS = 16 * SELECTOR + OFFSET`. So, when the IBM PC sets CS to 0xf000 and IP to 0xfff0, the memory address referenced is:

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

Now we can see that the PC starts executing 16 bytes from the end of the BIOS code. Therefore we shouldn't be surprised that the first thing that the BIOS does is `jmp` backwards; after all how much could it accomplish in just 16 bytes? When the BIOS runs, it sets up an interrupt descriptor table, initializes devices and searches for a bootable device (e.g., a floppy, hard drive, CDROM). When it finds a bootable devices, the BIOS loads the first sector (512 bytes) into memory at address `0x7c00 - 0x7d00` and sets CS:IP to `0000:7c00`. Like the BIOS load address, these addresses are fairly arbitrary--but they are fixed.

Side note: disks are by historical convention divided up in to 512 byte regions called sectors. This is the disk's minimum transfer granularity. Each read/write operation must be one or more sectors in size. If the disk is bootable, the first sector is also called the boot sector, since this is where the boot strapping code resides.

Set a break point at 0x7c00 and let the machine run until it hits the break point. Step through the code for five or ten instructions until you figure out what's going on.

Make sure you can answer these questions:

1. Disassemble `0x7c00`. Does it match the source file `ex1.S`?

2. Does the code match what you saw while single-stepping?

3. What is the machine code for a NOP?

4. What is the machine code that corresponds to the `jmp start` above? Write out the bytes in hex. Refer to the The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference. Explain each of the bytes.

5. You should have concluded that the last byte is a relative offset. What is its value in hex and in decimal? Refer to section 2.1 of the PC assembly book to learn how to convert the hex value, which is in 2s-complement representation, to a decimal value.

6. Is the relative offset of the `jmp` instruction relative to the instruction counter at the beginning or the end of the `jmp`? How do you know this?

Link vs. Load Address

The *load address* is the address at which a binary is loaded into memory. For example, the BIOS is loaded by the PC hardware at address 0xf0000. So this is the BIOS's load address.

Similarly, the BIOS loads the boot sector at address 0x7c00. So this is the boot sector's load address.

The *link address* is the address for which a binary is linked. Essentially, linking a binary for a given link address, prepares it to be loaded at that address. A program's link address in practice becomes subtly encoded within the binary in a multitude of ways, with the result that if a binary is not loaded at the address that it is linked for, things will not work.

In one sentence: the link address is the location where a binary assumes it is going to be loaded.

When our `Makefile` builds a disk image, it links the boot loader with a link address of `0x7c00`, matching the address where the BIOS will load it at run-time.

The file `ex2.S` illustrates the vague *"link address becomes subtly encoded in the binary"* claim above. The load versus link issue should be clear after working through the concrete example in `ex2.S`. Run `make ex2.disk` and restart `bochs`.

Make sure you can answer the following questions:

1. Trace the execution with Bochs' debugger. After EAX is loaded with with $here,

type 'info registers'. (You can also type 'dump_cpu' for a slightly more detailed view of the CPU's state.) What is the value of the EAX register?

2. Explain what 'jmp *%eax' does. What is the CS:IP before and after this instruction?

3. Explain why this code would not run correctly if the BIOS did not load it at address 0x7c00.

4. Explain why the code in `ex1.S` would still run correctly even if the BIOS did NOT load it at address 0x7c00.

5. Play with the `BOOTADDR` parameter in the `Makefile`. Explain how this value affects the value of EAX loaded by "movl $here,%eax". (Note that due to `ld` trying to align the code, you should only use multiples of 8 (i.e., hex numbers ending in 0 or 8) as your `BOOTADDR`. As a challenge, track down the behavior that occurs when `BOOTADDR` is not a multiple of 8.)

When object code contains no absolute addresses that ``subtly encode" the link address, we say that it is position-independent: it will behave the same no matter where it is loaded. Don't forget to reset `BOOTADDR` to `0x7c00` and rerun `make ex2.disk`.

Bochs Disk Images

Now it's time to examine the steps to build a bochs disk image closely. It is important to understand what is going on and why each step is important. For this exercise we'll be using the code `ex2.S` again (ie. you don't need to run make).

First, a handy tool for you to use is `xxd`. It dumps a binary file as hex:

```
[root@oslab root]$ xxd -g 1 -a disk
0000000: 90 90 90 90 66 b8 00 7c 00 00 66 ff e0 8d 74 00  ....f..|..f...t.
0000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0176ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The '*' means that every thing in `0000010-0176ff0` is identical.

Combine this with the source file `ex2.S` and perhaps single stepping through Bochs to answer the following questions:

1. Finish filling in this chart. It shows how the source code in `ex2.S` corresponds to the bytes in the `disk.img` file. The first to lines have been filled in for you.

```
instruction          machine code       byte start        length
 nop                    90               0             1
 nop                    90               1             1
 nop                     ?               ?             ?
 nop                     ?               ?             ?
 movl   $here,%eax        ?               ?             ?
 jmp    *%eax             ?               ?             ?
```

2. Play with the `BOOTADDR` in the `GNUmakefile` to see how the disk image changes. If

the BIOS loaded the disk image at a different address, could that ever affect the contents of the disk image? Why or why not? Now do you see why the link address should match the load address?

3. Now you should learn how a disk image is built by studying the output of make. Run `make ex2.disk`. Make sure you understand each step in the build process.

Don't forget to reset `BOOTADDR` to its original value.

Initializing the Stack

Read and understand the code in `ex3.S`--it gives an example of how to setup the stack. Type `make ex3.disk`.

To answer this question, you might like to use the 'x' command in the Bochs debugger. In the example below, we print out 10 words starting at address 0x7c00. And then we print out 16 bytes starting at 0x7c00.

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

```
<bochs:19> x /10w 0x7c00
[bochs]:
0x7c00 <bogus+0>:      0x8e66c031     0x00bc66d0     0x6600007c
0x000003e8
0x7c10 <bogus+16>:     0xfdeb6600     0x9090c366     0x00000000
0x00000000
0x7c20 <bogus+32>:     0x00000000     0x00000000
<bochs:20> x /16b 0x7c00
[bochs]:
0x7c00 <bogus+0>:      0x31    0xc0    0x66    0x8e    0xd0    0x66    0xbc
0x00
0x7c08 <bogus+8>:      0x7c    0x00    0x00    0x66    0xe8    0x03    0x00
0x00
<bochs:21>
```

Make sure you can answer these questions:

1. List the contents (after the BIOS executes) of 40 bytes of memory starting address 0x7c00. What do these bytes correspond to?

2. The "call subroutine" instruction pushes its return address on the stack. What is the value of SS:SP and the corresponding memory location right before and right after the "call"?

3. What value gets pushed on the stack? How is this value related to the load address? to the link address?

4. What does the "ret" instruction do? What is the value of SS:SP and the corresponding memory location right before and right after the "ret"?

5. Where in memory is the stack located versus the boot sector's code? Do they

overlap, abut, or are they spaced apart in memory?

6. How about versus the BIOS's memory location?

7. What would happen differently if `ex3` were loaded at a different place in memory?

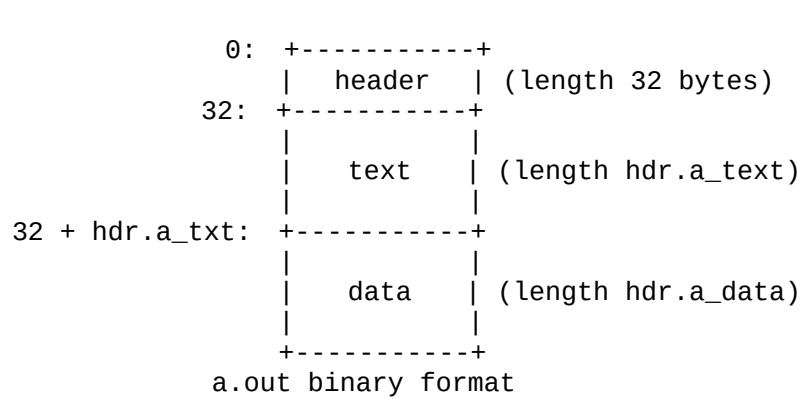
The boot loader

Finally we've made it to the boot loader!

The files `boot.S` and `bootc.c` contain a working boot loader. Your job is to first read and understand the code.

To make sense out of it you'll need to know what an a.out binary is. "a.out" is the traditional format of an executable program in Unix and Unix-like systems that is compiled and ready to run. The precise details of this format vary from one processor architecture and operating system to another, and in more recent versions of Linux and similar systems the a.out format has been largely replaced with ELF (Executable and Linkable Format), a substantially more powerful but also more complex executable program format. We're sticking to a.out format in this class for simplicity, since the GNU compiler toolchain still supports it.

An a.out binary starts with a header, followed by 2 sections: text and data. The text holds the program's machine code. The data section holds the programs (initialized) data. These section names obviously reflect the processor's viewpoint: anything that humans would consider "text", such as ASCII strings generated by the C compiler from string literals in the source code, will be found in the data section.




```
struct a_out_header {
    unsigned long  a_midmag;   /* flags<<26 | mid<<16 | magic */
    unsigned long  a_text;    /* text segment size */
    unsigned long  a_data;    /* initialized data size */
    unsigned long  a_bss;     /* uninitialized data size */
    unsigned long  a_syms;    /* symbol table size */
    unsigned long  a_entry;   /* entry point */
    unsigned long  a_trsize;  /* text relocation size */
    unsigned long  a_drsize;  /* data relocation size */
```

```
}
```
Of these fields, the bootloader only uses `a_text`, `a_data`, and `a_entry`. The meaning of the first two should be clear from the diagrams above. `a_entry` is the link address of the binary.

Using segmentation to work around position dependence

We've seen that in order to run position-dependent code, it must be loaded at its link address. `Gcc` uses position-dependent code to acess global variables or pass around function pointers. Thus, writing in C more-or-less implies having position-dependent code. Operating system kernels often link themselves at high virtual addresses (like 0xf0100020) so that user programs can use the lower part of the virtual address space. The reason for this arrangement will become clearer in the next lab. In order to run such a kernel, we need to load the kernel at that high address. Most machines don't even have that much memory (how much would it be?), so we'll use the segmentation hardware to make the kernel *appear* to be loaded there.

Run `make k0.disk` and then boot the disk in Bochs. The kernel that is booted has been carefully crafted to be position-independent even though it is written in C. It checks whether its load and link address match and prints a message accordingly. The kernel in `k0.disk` loads properly at 0x00100020.

On the other hand, if the boot loader were buggy, the kernel might behave like the one in `k1.disk`. Run `make k1.disk` and restart Bochs. (`K1.disk` simulates a buggy boot loader by writing the wrong load address into the a.out header.)

The kernel in `k2.disk` expects to be loaded at 0xf0100020 but is loaded at 0x00100020, because the boot loader only uses the bottom 24 bits of the entry point as the load address. In this case, the kernel gets confused and prints garbage. Run `make k2.disk` and restart Bochs to see this. This means the kernel has loaded incorrectly.

(**Challenge!** Fix the kernel to work in all situations.)

We can fake the top 8 bits of the entry point using the segmentation hardware. That is, we want to set up the kernel code segment so that the kernel entry point in that segment corresponds to the actual physical location where we're already loading the kernel. Eventually the kernel will set up page tables to accomplish the same thing, but using the segmentation hardware will do for now.

You task is to configure the segmentation hardware to provide the illusion that the kernel was loaded at its link address. Edit `bootc.c` to set up the segment descriptor `kdesc` properly to construct the illusion. You might want to consult section 3.4 of IA-32 Intel Architecture Software Developer's Manual, Volume 3: System programming guide. The solution does not require using any esoteric features, just basic segmentation, though in a non-obvious way.

Once you have `bootc.c` set up properly, you should be able to `make k2.disk` and then boot Bochs and get a successful result from the kernel.

Sadly, it is easy to make Bochs fall over while doing this exercise. If Bochs crashes, you might try again with "trace-on" so that at least you can figure out which instruction it was trying to simulate when it crashed. If Bochs falls over, it means your code is buggy - a real PC would have rebooted.

Challenge! Read and understand `kernel.c`.

Part 2: GCC Calling Conventions

In the second part of this lab we'll implement the standard C library routines `setjmp` and `longjmp`, declared thus:

```
int  setjmp(jmp_buf env);
void longjmp(jmp_buf env, int);
```

`Setjmp` saves its stack environment in env for later use by `longjmp`. It returns 0.

`Longjmp` restores the environment saved by the last call of `setjmp`.It then causes execution to continue as if the call of `setjmp` had just returned with value val. The invoker of `setjmp` must not itself have returned in the interim. All accessible data have values as of the time `longjmp` was called.

One use for `setjmp/longjmp` is as a "poor man's" exception handling mechanism, since C does not have "try/catch" exceptions like C++ and Java do. These functions can also be used in more sophisticated ways, however, such as to switch between several entirely different execution stacks on the fly; user-level multithreading packages such as "pthreads" frequently do this.

Since the calling convention is the same in our a.out gcc as it is in the standard Linux x86 ELF gcc, we'll use the Linux gcc to make debugging easier. (Debugging is always easier when it's done outside the kernel.)

To understand how these routines behave, run the setjmp-example.c program. Make sure you understand why you see each line of output. Just as important, make sure you understand why you do *not* see so many of the ``exiting" messages in the output.

To avoid conflicting with the C library definitions, our routines will be called `setjmp828` and `longjmp828` and declared thus:

```
typedef struct stackenv stackenv_t;
struct stackenv
{
   uint32_t ebp;
   uint32_t ebx;
   uint32_t esi;
   uint32_t edi;
   uint32_t esp;
   uint32_t ret;
};
int  setjmp828(stackenv_t *env);
void longjmp828(stackenv_t env, int);
```

(Our declarations also make it explicit that `env` is a pointer to a structure. The C library

declarations leave this implicit by typedefing `jmp_buf` to be an array of integers.)

Since they deal with low-level register manipulations, `setlabel` and `gotolabel` must be implemented in assembly. Your job is to implement `setjmp828` and `longjmp828` in `lab1/jmp/jmp.S`. An outline has been written for you.

You can test your implementation by running `make grade`. Note that if the grading program seg faults, it's probably your bug.

Make sure you can answer the following questions:

1. Why doesn't `setjmp` need to save EAX, ECX, or EDX?

2. Why does the `setjmp` definition include the requirement ``The invoker of `setjmp` must not itself have returned in the interim."? (In other words, what's wrong with the setjmp-bad.c program.? Note that even though the code appears to work, it's still wrong.)

   Challenge! Read and understand `testjmp.S` and `testjmpc.c`.

   This completes the lab.