Jos Lab 1: Booting a PC

Yuan Xue@PPI

Hand out: Dec. 7th

Deadline: Dec. 21th 14:00 (GMT+8) No Extension

UPDATE: Please be sure to get the latest version using **svn**.

Introduction

This lab is split into three parts. The first part concentrates on getting familiarized with x86 assembly language, the QEMU x86 emulator, and the PC's power-on bootstrap procedure. The second part examines the boot loader for our course kernel, which resides in the `boot` directory of the `lab` tree. Finally, the third part delves into the initial template for our course kernel itself, named JOS, which resides in the `kernel` directory.

Software Setup

Since in the this and subseqent labs, we will use Git as the version control system, highly recommend learning how to use Git.

To get the lab in your PC, you need to *clone* the course repository, by running the commands below. You must use an x86 architecture; that is, `uname -a` should mention `i386 GNU/Linux` or `i686 GNU/Linux`.

You can use the virtual machine provided by us. Virtual Machine

```
shell%    svn   --username=StuID checkout svn://10.176.34.18/xueyuan_jos1_repo/StuID
shell%    cd    jos01
...
shell% tar zvxf jos-2019-spring.tar.gz
```

Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you can *commit* your changes by running:

```
shell%  svn add handin
shell%. svn commit -m "handin jos01
```

To help you set up running environment, tools page has directions on how to set up `gcc` `gdb` and `QEMU` for use with JOS. We also have set up the appropriate compilers and simulators for you in the course supplied Debian VMware image.

Grading and Hand-In Procedure

Grading: coding**(70%)**, document**(15%)**, interview**(15%)**.

When you are ready to hand in your lab, run `make handin` in the your `lab` directory. This will make a tar file for you, which you can then upload to the ta's public ftp, before you hand in your source code, you should rename the tar file with your student id, the name of the final file you turnin should be **{your student id}.tar.gz**. `make handin` provides more specific directions.

Generally speaking, you can **ONLY** submit your code once to our ftp, because you cannot modify or replace any files already existing on our server. However, there may have been some emergencies which you need to improve your handin. Please add a version number with '_' after your student id, e.g. 123456_1.tar.gz, and submit again.

Keep the submitted file carefully since you might need it to check your grades.

For the lab1, you need to hand in your souce code and a document to describe the design of lab1(**The more detail, the more credits you may get!**). Note that you should name the document "lab1.pdf" and add it in **joslab1/** directory:

You do not need to turn in answers to any of the questions in the text of the lab. But we will ask the questions about them in the interview, and we believe that they will help with the rest of the lab.

We will be grading your solutions with a grading program. You can run `make grade` to test your solutions with the grading program.

Part 1: PC Bootstrap

The purpose of the first exercise is to introduce you to x86 assembly language and the PC bootstrap process, and to get you started with QEMU and QEMU/GDB debugging. You will not have to write any code for this part of the lab, but you should go through it anyway for your own understanding and be prepared to answer the questions posed below.

Getting Started with x86 assembly

If you are not already familiar with x86 assembly language, you will quickly become familiar with it during this course! The PC Assembly Language Book is an excellent place to start. Hopefully, the book contains mixture of new and old material for you.

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 Brennan's Guide to Inline Assembly.

Exercise 1. Read or at least carefully scan the entire PC Assembly Language book, except that you should skip all sections after 1.3.5 in chapter 1, which talk about features of the NASM assembler that do not apply directly to the GNU assembler. You may also skip chapters 5 and 6, and all sections under 7.2, which deal with processor and language features we won't use in JOS.

Also read the section "The Syntax" in Brennan's Guide to Inline Assembly to familiarize yourself with the most important features of GNU assembler syntax.

Certainly the definitive reference for x86 assembly language programming is Intel's instruction set architecture reference, which you can find on the reference page in two flavors: an HTML edition of the old 80386 Programmer's Reference Manual, which is much shorter and easier to navigate than more recent manuals but describes all of the x86 processor features that we will make use of in JOS; and the full, latest and greatest IA-32 Intel Architecture Software Developer's Manuals from Intel, covering all the features of the most recent processors that we won't need in class but you may be interested in learning about. An equivalent (but even longer) set of manuals is available from AMD, which also covers the new 64-bit extensions now appearing in both AMD and Intel processors.

You should skim the recommended chapters of the PC Assembly book, and "The Syntax" section in Brennan's Guide now. Save the Intel/AMD architecture manuals for later or use them for reference when you want to look up the definitive explanation of a particular processor feature or instruction.

Simulating the x86

Instead of developing the operating system on a real, physical personal computer (PC), we use a simulator, which emulates a complete PC faithfully: the code you write for the simulator will boot on a real PC too. Using a simulator simplifies debugging; you can, for example, set break points inside of the simulated x86, which is difficult to do with the silicon-version of an x86.

In JOS we will use the QEMU Emulator, a modern and relatively fast emulator. While QEMU's built-in monitor provides only limited debugging support, QEMU can act as a remote debugging target for the GNU debugger (GDB), which we'll use in this lab to step through the early boot process.

To get started, checkout the Lab 1 files into your own directory as described above in "Software Setup", then type `make` in the `lab` directory to build the minimal JOS boot loader and kernel you will start with. (It's a little generous to call the code we're running here a "kernel," but we'll flesh it out throughout the semester.)

```
shell% cd jos-2019-spring
shell% make
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
```

(If you get errors like `undefined reference to `__udivdi3'`, you are probably on x86_64 Linux and don't have the 32-bit gcc multilib. If you're running Debian or Ubuntu, try installing the gcc-multilib package.)

Now you're ready to run QEMU, supplying the file `obj/kern/kernel.img`, created above, as the contents of the emulated PC's "virtual hard disk." This hard disk image contains both our boot loader (`obj/boot/boot`) and our kernel (`obj/kern/kernel`).

```
shell% make qemu # or "make qemu-nox"
```

This executes QEMU with the options required to set the hard disk and direct serial port output to the terminal. Some text should appear in the QEMU window:

```
Booting from Hard Disk...
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Everything after '`Booting from Hard Disk...`' was printed by our skeletal JOS kernel; the `K>` is the prompt printed by the small *monitor*, or interactive control program, that we've included in the kernel. These lines printed by the kernel will also appear in the regular shell window from which you ran QEMU. This is because for testing and lab grading purposes we have set up the JOS kernel to write its console output not only to the virtual VGA display (as seen in the QEMU window), but also to the simulated PC's virtual parallel port, which QEMU outputs to its own standard output because of the -serial argument. Likewise, the JOS kernel will take input from both the keyboard and the serial port, so you can give it commands in either the VGA display window or the terminal running QEMU. Alternatively, you can use the serial console without the virtual VGA by running **`make qemu-nox`**.

There are only two commands you can give to the kernel monitor, `help` and `kerninfo`.

```
K> help
help - display this list of commands
kerninfo - display information about the kernel
K> kerninfo
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101871 (virt)  00101871 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112940 (virt)  00112940 (phys)
Kernel executable memory footprint: 75KB
K>
```

The `help` command is obvious, and we will shortly discuss the meaning of what the `kerninfo` command prints. Although simple, it's important to note that this kernel monitor is running "directly" on the "raw (virtual) hardware" of the simulated PC. This means that you should be able to copy the contents of `obj/kern/kernel.img` onto the first few sectors of a *real* hard disk, insert that hard disk into a real PC, turn it on, and see exactly the same thing on the PC's real screen as you did above in the QEMU window. (We don't recommend you do this on a real machine with useful information on its hard disk, though, because copying `kernel.img` onto the beginning of its hard disk will trash the master boot record and the beginning of the first partition, effectively causing everything previously on the hard disk to be lost!)

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 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.

Recent x86 processors can support *more* than 4GB of physical RAM, so RAM can extend further above 0xFFFFFFFF. In this case the BIOS must arrange to leave a *second* hole in the system's RAM at the top of the 32-bit addressable region, to leave room for these 32-bit devices to be mapped. Because of design limitations JOS will use only the first 256MB of a PC's physical memory anyway, so for now we will pretend that all PCs have "only" a 32-bit physical address space. But 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

In this portion of the lab, you'll use QEMU's debugging facilities to investigate how an IA-32 compatible computer boots. Open two terminal windows. In one, enter **make qemu-gdb # or "make qemu-nox-gdb"**. This starts up QEMU, but QEMU stops just before the processor executes the first instruction and waits for a debugging connection from GDB. In the second terminal, from the same directory you ran **make**, run **gdb**. You should see something like this,

```
K> make gdb
GNU gdb (GDB) 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
(gdb)
```

We provided a `.gdbinit` file that set up GDB to debug the 16-bit code used during early boot and directed it to attach to the listening QEMU.

The following line:

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

is GDB's disassembly of the first instruction to be executed. 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?

Exercise 2. 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 Jos reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

When the BIOS runs, it sets up an interrupt descriptor table and initializes various devices such as the VGA display. This is where the "`Starting SeaBios`" 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.

Part 2: 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 JOS, however, we will use the conventional hard drive boot mechanism, which means that our boot loader must fit into a measly 512 bytes. The boot loader consists of one assembly language source file, `boot/boot.S`, and one C source file, `boot/main.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 JOS 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 `obj/boot/boot.asm`. This file is a disassembly of the boot loader that our GNUmakefile 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 breakpoints 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.

To examine instructions in memory (besides the immediate next one 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.

Exercise 3. Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.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 GNU disassembly in `obj/boot/boot.asm` and the GDB

Trace into `bootmain()` in `boot/main.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?

Loading the Kernel

We will now look in further detail at the C language portion of the boot loader, in `boot/main.c`. But before doing so, this is a good time to stop and review some of the basics of C programming.

Exercise 4. 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'). We recommend that students purchase this book (here is an Amazon Link).

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for 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 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, though not as strongly recommended. A tutorial by Ted Jensen that cites K&R heavily is available in the course readings.

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 in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

To make sense out of `boot/main.c` you'll need to know what an ELF binary is. When you compile and link a C program such as the JOS 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 `obj/kern/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 in this class.

For purposes of JOS, 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:

```
shell% i386-jos-elf-objdump -h obj/kern/kernel
```

You can substitute `objdump` for `i386-jos-elf-objdump` if your computer uses an ELF toolchain by default like most modern Linuxen and BSDs.

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:

```
shell% i386-jos-elf-objdump -f obj/kern/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).

Exercise 5. 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.)

Link vs. Load Address

The *load address* of a binary is the memory address at which a binary is *actually* loaded. 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* of a binary is the memory address for which the binary is linked. Linking a binary for a given link address prepares it to be loaded at that address. The linker encodes the link address in the binary in various ways, for example when the code needs the address of a global variable, with the result that a binary usually won't work if it is not loaded at the address that it is linked for.

In one sentence: the link address is the location where a binary *assumes* it is going to be loaded, while the load address is the location where a binary *is* loaded. It's up to us to make sure that they turn out to be the same.

Look at the `-Ttext` linker command in `boot/Makefrag`, and at the address mentioned early in the linker script in `kern/kernel.ld`. These set the link address for the boot loader and kernel respectively.

Exercise 6. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in `boot/Makefrag` to something wrong, run `make clean`, recompile the lab with `make`, and trace into the boot loader again to see what happens. Don't forget to change the link address back and `make clean` afterwards!

When object code contains no absolute addresses that encode the link address in this fashion, we say that the code is *position-independent*: it will behave correctly no matter where it is loaded. GCC can generate position-independent code using the `-fpic` option, and this feature is used extensively in modern shared libraries that use the ELF executable format. Position independence typically has some performance cost, however, because it restricts the ways in which the compiler may choose instructions to access the program's data. We will not use `-fpic` in JOS.

Part 3: The Kernel

We will now start to examine the minimal JOS kernel in a bit more detail. (And you will finally get to write some code!) Like the boot loader, the kernel begins with some assembly language code that sets things up so that C language code can execute properly.

Using virtual memory to work around position dependence

When you inspected the boot loader's link and load addresses above, they matched perfectly, but there was a (rather large) disparity between the *kernel's* link address (as printed by objdump) and its load address. Go back and check both and make sure you can see what we're talking about. (Linking the kernel is more complicated than the boot loader, so the link and load addresses are at the top of `kern/kernel.ld`.)

Operating system kernels often like to be linked and run at very high *virtual address*, such as 0xf0100000, in order to leave the lower part of the processor's virtual address space for user programs to use. The reason for this arrangement will become clearer in the next lab.

Many machines don't have any physical memory at address 0xf0100000, so we can't count on being able to store the kernel there. Instead, we will use the processor's memory management hardware to map virtual address 0xf0100000 (the link address at which the kernel code *expects* to run) to physical address 0x00100000 (where the boot loader loaded the kernel into physical memory). This way, although the kernel's virtual address is high enough to leave plenty of address space for user processes, it will be loaded in physical memory at the 1MB point in the PC's RAM, just above the BIOS ROM. This approach requires that the PC have at least a few megabytes of physical memory (so that physical address 0x00100000 works), but this is likely to be true of any PC built after about 1990.

In fact, in the next lab, we will map the *entire* bottom 256MB of the PC's physical address space, from physical addresses 0x00000000 through 0x0fffffff, to virtual addresses 0xf0000000 through 0xffffffff respectively. You should now see why JOS can only use the first 256MB of physical memory.

For now, we'll just map the first 4MB of physical memory, which will be enough to get us up and running. We do this using the hand-written, statically-initialized page directory and page table in `kern/entrypgdir.c`. For now, you don't have to understand the details of how this works, just the effect that it accomplishes. Up until `kern/entry.S` sets the `CR0_PG` flag, memory references are treated as physical addresses (strictly speaking, they're linear addresses, but boot/boot.S set up an identity mapping from linear addresses to physical addresses and we're never going to change that). Once `CR0_PG` is set, memory references are virtual addresses that get translated by the virtual memory hardware to physical addresses. `entry_pgdir` translates virtual addresses in the range 0xf0000000 through 0xf0400000 to physical addresses 0x00000000 through 0x00400000, as well as virtual addresses 0x00000000 through 0x00400000 to physical addresses 0x00000000 through 0x00400000. Any virtual address that is not in one of these two ranges will cause a hardware exception which, since we haven't set up interrupt handling yet, will cause QEMU to dump the machine state and exit (or endlessly reboot if you aren't using the 6.828-patched version of QEMU).

The x86 processor has two distinct memory management mechanisms that JOS could use to achieve this mapping: *segmentation* and *paging*. Both are described in full detail in the 80386 Programmer's Reference Manual and the IA-32 Developer's Manual, Volume 3. When the JOS kernel first starts up, it initially uses segmentation to establish the desired virtual-to-physical mapping, because it is quick and easy - and the x86 processor requires us to set up the segmentation hardware in any case, because it can't be disabled!

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.

What is the first instruction *after* the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in `kern/entry.S`, trace into it, and see if you were right.

Formatted Printing to the Console

Most people take functions like `printf()` for granted, sometimes even thinking of them as "primitives" of the C language. But in an OS kernel, we have to implement all I/O ourselves.

Read through `kern/printf.c`, `lib/printfmt.c`,and `kern/console.c`, and make sure you understand their relationship.It will become clear in later labs why `printfmt.c` is located in the separate `lib` directory.

Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment. Remember the octal number should begin with '0'.

Exercise 9. You need also to add support for the "+" flag, which forces to precede the result with a plus or minus sign (+ or -) even for positive numbers.

Be able to answer the following questions:

1. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?
2. Explain the following from `console.c`:

```
	if (crt_pos >= CRT_SIZE) {
		int i;
		memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
		for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
			crt_buf[i] = 0x0700 | ' ';
		crt_pos -= CRT_COLS;
	}
```

3. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.
Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

   - In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?
   - List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `vcprintf` list the values of its two arguments.

4. Run the following code.

```
unsigned int i = 0x00646c72;
    cprintf("H%x Wo%s", 57616, &i);
```

What is the output? Explain how this output is arrived out in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

Here's a description of little- and big-endian and a more whimsical description.

5. In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
    cprintf("x=%d y=%d", 3);
```

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

Exercise 10. Enhance the cprintf function to allow it print with the %n specifier, you can consult the %n specifier specification of the C99 printf function for your reference by typing "man 3 printf" on the console. In this lab, we will use the **char *** type argument instead of the C99 **int *** argument, that is, "the number of characters written so far is stored into the **signed char** type integer indicated by the char * pointer argument. No argument is converted." You must deal with some special cases properly, because we are in kernel, such as when the argument is a NULL pointer, or when the char integer pointed by the argument has been overflowed. Find and fill in this code fragment.

Exercise 11. Modify the function `printnum()` in `lib/printfmt.c` to support `"%-"` when printing numbers. With the directives starting with "%-", the printed number should be left adjusted. (i.e., paddings are on the right side.) For example, the following function call:

```
cprintf("test:[%-5d]", 3)
```

, should give a result as

```
"test:[3    ]"
```

(4 spaces after '3'). Before modifying `printnum()`, make sure you know what happened in function `vprintffmt()`.

The Stack

In the final exercise of this lab, we will explore in more detail the way the C language uses the stack on the x86, and in the process write a useful new kernel monitor function that prints a *backtrace* of the stack: a list of the saved Instruction Pointer (IP) values from the nested `call` instructions that led to the current point of execution.

Exercise 12. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

The x86 stack pointer (`esp` register) points to the lowest location on the stack that is currently in use. Everything *below* that location in the region reserved for the stack is free. Pushing a value onto the stack involves decreasing the stack pointer and then writing the value to the place the stack pointer points to. Popping a value from the stack involves reading the value the stack pointer points to and then increasing the stack pointer. In 32-bit mode, the stack can only hold 32-bit values, and esp is always divisible by four. Various x86 instructions, such as `call`, are "hard-wired" to use the stack pointer register.

The `ebp` (base pointer) register, in contrast, is associated with the stack primarily by software convention. On entry to a C function, the function's *prologue* code normally saves the previous function's base pointer by pushing it onto the stack, and then copies the current `esp` value into `ebp` for the duration of the function. If all the functions in a program obey this convention, then at any given point during the program's execution, it is possible to trace back through the stack by following the chain of saved `ebp` pointers and determining exactly what nested sequence of function calls caused this particular point in the program to be reached. This capability can be particularly useful, for example, when a particular function causes an `assert` failure or `panic` because bad arguments were passed to it, but you aren't sure *who* passed the bad arguments. A stack backtrace lets you find the offending function.

Exercise 13. To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

The above exercise should give you the information you need to implement a stack backtrace function, which you should call `mon_backtrace()`. A prototype for this function is already waiting for you in `kern/monitor.c`. You can do it entirely in C, but you may find the `read_ebp()` function in `inc/x86.h` useful. You'll also have to hook this new function into the kernel monitor's command list so that it can be invoked interactively by the user.

The backtrace function should display a listing of function call frames in the following format:

```
Stack backtrace:
  eip f0100a62  ebp f0109e58  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  eip f01000d6  ebp f0109ed8  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

The first line printed reflects the *currently executing* function, namely `mon_backtrace` itself, the second line reflects the function that called `mon_backtrace`, the third line reflects the function that called that one, and so on. You should print *all* the outstanding stack frames. By studying `kern/entry.S` you'll find that there is an easy way to tell when to stop.

Within each line, the `ebp` value indicates the base pointer into the stack used by that function: i.e., the position of the stack pointer just after the function was entered and the function prologue code set up the base pointer. The listed `eip` value is the function's *return instruction pointer*: the instruction address to which control will return when the function returns. The return instruction pointer typically points to the instruction after the `call` instruction (why?). Finally, the five hex values listed after `args` are the first five arguments to the function in question, which would have been pushed on the stack just before the function was called. If the function was called with fewer than five arguments, of course, then not all five of these values will be useful. (Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?)

Here are a few specific points you read about in K&R Chapter 5 that are worth remembering for the following exercise and for future labs.

- If `int *p = (int*)100`, then `(int)p + 1` and `(int)(p + 1)` are different numbers: the first is `101` but the second is `104`. When adding an integer to a pointer, as in the second case, the integer is implicitly multiplied by the size of the object the pointer points to.
- `p[i]` is defined to be the same as `*(p+i)`, referring to the i'th object in the memory pointed to by p. The above rule for addition helps this definition work when the objects are larger than one byte.

- `&p[i]` is the same as `(p+i)`, yielding the address of the i'th object in the memory pointed to by p.

Although most C programs never need to cast between pointers and integers, operating systems frequently do. Whenever you see an addition involving a memory address, ask yourself whether it is an integer addition or pointer addition and make sure the value being added is appropriately multiplied or not.

Exercise 14. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run **make grade** to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

At this point, your backtrace function should give you the addresses of the function callers on the stack that lead to `mon_backtrace()` being executed. However, in practice you often want to know the function names corresponding to those addresses. For instance, you may want to know which functions could contain a bug that's causing your kernel to crash.

To help you implement this functionality, we have provided the function `debuginfo_eip()`, which looks up `eip` in the symbol table and returns the debugging information for that address. This function is defined in `kern/kdebug.c`.

Exercise 15. Modify your stack backtrace function to display, for each `eip`, the function name, source file name, and line number corresponding to that `eip`.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- run **i386-jos-elf-objdump -h obj/kern/kernel**
- run **i386-jos-elf-objdump -G obj/kern/kernel**
- run **i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c**, and look at init.s.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Add a `backtrace` command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
  eip f01008ae  ebp f010ff78  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143 monitor+106
  eip f0100193  ebp f010ffd8  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49 i386_init+59
  eip f010003d  ebp f010fff8  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70 <unknown>+0
K>
```

Each line gives the file name and line within that file of the stack frame's `eip`, followed by the name of the function and the offset of the `eip` from the first instruction of the function (e.g., `monitor+106` means the return `eip` is 106 bytes past the beginning of `monitor`).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

You may find that the some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUMakefile`, the backtraces may make more sense (but your kernel will run more slowly).

Exercise 16. Recall the buffer overflow attack in ICS Lab. Modify your start_overflow function to use a technique similar to the buffer overflow to invoke the do_overflow function. **You must use the above cprintf function with the %n specifier you augmented in "Exercise 10" to do this job, or else you won't get the points of this exercise**, and the do_overflow function should return normally.

Exercise 17. There is a "time" command in Linux. The command counts the program's running time.

```
$time ls
a.file b.file ...

real    0m0.002s
user    0m0.001s
sys     0m0.001s
```

In this exercise, you need to implement a rather easy "time" command. The output of the "time" is the running time (in clocks cycles) of the command. The usage of this command is like this: "time [command]".

```
K> time kerninfo
Special kernel symbols:
 _start f010000c (virt)  0010000c (phys)
 etext  f0101a75 (virt)  00101a75 (phys)
 edata  f010f320 (virt)  0010f320 (phys)
 end    f010f980 (virt)  0010f980 (phys)
Kernel executable memory footprint: 63KB
kerninfo cycles: 23199409
K>
```

Here, 23199409 is the running time of the program in cycles. As JOS has no support for time system, we could use CPU time stamp counter to measure the time.

Hint: You can refer to instruction "rdtsc" in Intel Mannual for measuring time stamp. ("rdtsc" may not be very accurate in virtual machine environment. But it's not a problem in this exercise.)

This completes the lab. Type `make grade` in the `lab` directory for test, then type **`make handin`** to pack the files, rename the lab1-handin.tar.gz file to **{your student id}.tar.gz**, and follow the directions for uploading your lab tar file onto ta's ftp.