

Lab 0: Environment tutorial

Credit: Tutorial component based on original MIT 6.828 Lab 1, available online [here](#).

This is an Optional Lab. I still recommend that you do it to understand how an OS fits within a computing system -- It could also make life easier in the future :-). However, be warned that most of you will find that some parts require knowledge and ideas that you may not know. Still, its ok to skim over these parts and try to get the overall gist of the activities in the lab.

Introduction

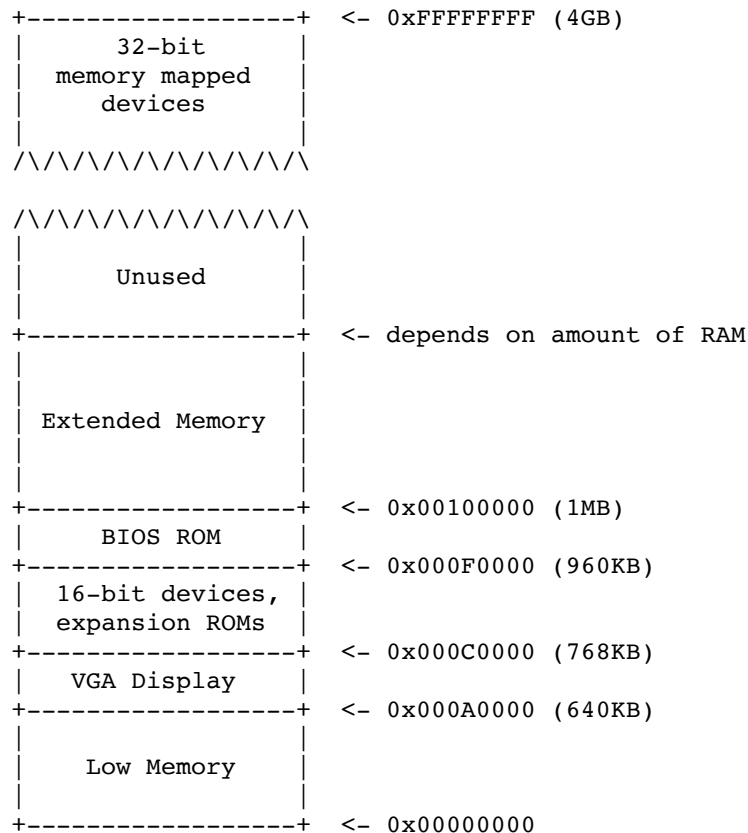
Part 0: Setup and getting started QEMU and gdb are already installed on sledge.

Please follow the setup instructions on the class webpage to get xv6 downloaded, installed and working with qemu and gdb.

QEMU is an open source emulator for complete PC systems.

Part 1: The PC's Physical Address Space

We will start with some detail about how a PC starts up. A PC's physical address space is hard-wired to have the following general layout:



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.

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 of xv6 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. QEMU is virtual machine software (similar to Xen, KVM, virtualBox, etc...); it allows each of us to get their own private hardware to boot an OS on.

nox: no user interface

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`. If you have problems, make sure that you have followed the instructions for installing and running xv6. Note that the order of how you start these matters. You should see something like this,


```
sledge% 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 ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

The terminal where qemu was run will be hanged after you run gdb. So, to quit, first go to the gdb tab and quit gdb, then on the same console, run "kill \$(pgrep qemu)". This will kill qemu.

Before we dive in, I have to warn you that understanding the following in detail requires that you know a little bit about virtual memory and segmentation -- topics that I do not expect you to know yet. However, it's ok if you do not understand the details. Virtual addresses are addresses that the program uses and that are produced by the compiler. These are translated into physical addresses to get the address of the data in the actual physical memory. When the PC boots, it is in real mode for a few instructions where the code uses physical addresses directly and no translation is necessary.

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. (If it doesn't work, you may have to add an `add-auto-load-safe-path` in your `.gdbinit` in your home directory to convince `gdb` to process the `.gdbinit` we provided. `gdb` will tell you if you have to do this.)

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 = 0xffff0`. **CS is the current segment register, and IP is the instruction pointer, or Program Counter.**
- 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 0xffff0, 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 0xffff0, the physical address referenced is:

```
16 * 0xf000 + 0xffff0    # in hex multiplication by 16 is
= 0xf0000 + 0xffff0      # easy--just append a 0.
= 0xfffff0
```

0xfffff0 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?

Once you added breakpoint in gdb, you can run "layout src" to give a GUI of the current execution files.

Exercise 1 (optional). 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. For a quick summary of gdb's commands (and especially the assembly level debugging, please take a look at [gdb assembly quick start](#) page. You might want to look at [Phil Storrs I/O Ports Description](#), as well as other materials on the [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" message you see in the QEMU window comes 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 this class, 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, `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. 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 xv6 resources 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.

You can set address 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 2.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `bootasm.s`, using the source code. You can also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader.

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.

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?
- Where is the first instruction of the kernel?
- 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 `bootmain.c`. But before doing so, this is a good time to stop and review some of the basics of C programming.

Warning: 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. We will have some support in the lab sessions to make sure that we give you a refresher on pointers if you need it.

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 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 `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](#) 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. The [Wikipedia page](#) has a short description.

For our 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.

Examine the full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:

```
sledge% objdump -h kernel
```

If your computer does not use an ELF toolchain by default (e.g., a MacOS) like most modern Linuxes and BSDs, you need to download the cross compiler tools and use the appropriate `objdump` binary.

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.

Take particular note of the "VMA" (or *link address*) and the "LMA" (or *load address*) of the `.text` section. The load address of a section is the memory address at which that section should be loaded into memory.

The link address of a section is the memory address from which the section expects to execute. The linker encodes the link address in the binary in various ways, such as when the code needs the address of a global variable, with the result that a binary usually won't work if it is executing from an address that it is not linked for. (It is possible to generate *position-independent* code that does not contain any such absolute addresses. This is used extensively by modern shared libraries, but it has performance and complexity costs, so we won't be using it)

Typically, the link and load addresses are the same. The boot loader uses the ELF *program headers* to decide how to load the sections. The program headers specify which parts of the ELF object to load into memory and the destination address each should occupy. You can inspect the program headers by typing:


```
sledge% objdump -x kernel
```

The program headers are then listed under "Program Headers" in the output of `objdump`. The areas of the ELF object that need to be loaded into memory are those that are marked as "LOAD". Other information for each program header is given, such as the virtual address ("vaddr"), the physical address ("paddr"), and the size of the loaded area ("memsz" and "filesz").

Back in `bootmain.c`, the `ph->p_pa` field of each program header contains the segment's destination physical address (in this case, it really is a physical address, though the ELF specification is vague on the actual meaning of this field).

The BIOS loads the boot sector into memory starting at address `0x7c00`, so this is the boot sector's load address. This is also where the boot sector executes from, so this is also its link address. We set the link address by passing `-Ttext 0x7c00` to the linker in the `Makefile`, so the linker will produce the correct memory addresses in the generated code.

Exercise. 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 `Makefile` 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` again afterward!

Look back at the load and link addresses for the kernel. Unlike the boot loader, these two addresses aren't the same: the kernel is telling the boot loader to load it into memory at a low address (1 megabyte), but it expects to execute from a high address. We'll dig in to how we make this work in the next section.

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:

```
sledge% objdump -f kernel
```

You should now be able to understand the minimal ELF loader in `bootmain.c`. It reads each section of the kernel from disk into memory at the section's load address and then jumps to the kernel's entry point.

Exercise 6. We can examine memory using GDB's `x` command. The [GDB manual](#) has full details, but for now, it is enough to know that the command `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).

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.)

Part 3: The Kernel

We will now start to examine the JOS kernel in a bit more detail. 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`, or `0x80100000` in order to leave the lower part of the processor's virtual address space for user programs to use.

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.

Later, we will map the *entire* bottom 256MB of the PC's physical address space, from physical addresses 0x00000000 through 0x0ffffff, 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 at the bottom of `main.c`. For now, you don't have to understand the details of how this works, just the effect that it accomplishes. Up until `entry.S` sets the `CR0_PG` flag, memory references are treated as physical addresses (strictly speaking, they're linear addresses, but `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).

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in `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 `printf.c`, `printfmt.c`, and `console.c`, and make sure you understand their relationship.

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`:

```

1      if (crt_pos >= CRT_SIZE) {
2          int i;
3          memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4          for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5              crt_buf[i] = 0x0700 | ' ';
6          crt_pos -= CRT_COLS;
7      }
```

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](#).

3. 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);
```

4. 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?

Challenge Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret [ANSI escape sequences](#) embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on the 6.828 reference page, and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

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 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.

This completes the lab. which is optional