Project 1

Due: TBA

Last updated: 2025/05/26 15:37:14

Update history:

```
2025/05/26: initial version
```

---

1. Introduction
2. Overview
3. Assignment Details
4. Turning In Your Solution
5. Future Work

---

1. Introduction

The primary goal of this assignment is to familiarize yourself with the boot procedure for x86-based processors. To this end, you will boot up a simple OS using the QEMU x86 Emulator, and will use GDB to examine the code and registers as the system boots and executes.

The source code for the OS is found on the Ubuntu® systems. This is also where your submission for this assignment is to be made.

---

2. Overview

Xv6 is a teaching operating system developed in the summer of 2006 for MIT's operating systems course, 6.828: Operating System Engineering. It comes with a simple build system, code for some useful user-level utilities that can be run from within xv6, and of course code for the operating system itself.

In the sections that follow, you will find a series of questions and exercises enclosed in boxed paragraphs:

Questions involve reading and analyzing source code, or to try “tweaking” something in the kernel source code, or something similar.

Exercises involve adding code to the kernel to add functionality.

Challenges are things you could consider as possible modifications to the kernel. They are “thought experiments” and are not work to be done for this assignment.

Please write up and hand in answers to the questions along with C source files containing any code you develop for the exercises. Put the answers to questions in a file named `answers.txt` and submit this along with your C code using the command shown in the *Turning In Your Solution* section (below).

You will also find some paragraphs with thin black borders. These contain information that may be particularly important, or they may point you toward other reference material that could help explain architectural or coding details.

```
Some boxed paragraphs may look like this. These are typically "screenshot"
paragraphs, showing you what you may expect to see in a shell window as you
perform some tasks. They use a fixed-width font similar to the one found in
most terminal emulators.
```

You may do this assignment individually, or together, as you prefer. While there is some benefit to working alone on a project such as this, working together gives you the opportunity to “brainstorm” when looking at strange code or when you run into problems as you modify the code base.

At the top of your `answers.txt` file, place the following text:

```
CSCI-352 Project 1
Submitted by:  YOUR_NAMES_HERE
```

Replace `YOUR_NAMES_HERE` with your name(s).

At the end of this writeup is a section entitled *Future Work* which contains several “challenge” questions. These are *not* to be turned in; rather, they are to get you thinking about other ways the basic system could be modified or enhanced.

The tools you will need for this assignment are straightforward, and are all available on the CS lab systems:

- Standard compilation and linking tools: `gcc`, `ld`, `as`.
  Our main-net machines have GCC 9.4.0 and GNU Binutils 2.34 installed, and the project 1 code has been tested using these tools, but any recent version should work. (*Note:* other C compilers and linkers may not work, as the source code makes use of “GNU-isms” in many places.)
- A working version of `gdb`.
  Our systems have GDB version 10.2 installed.
- Tools for working with object files: `nm`, `objcopy`, `objdump`.
- A working version of QEMU.
  See below for information about QEMU versions.
- Miscellaneous other system tools: `nm`, `dd`, `perl`.

Our lab machines have GCC 9.4.0 (`gcc`), GNU Binutils 2.34 (`ld`, `as`, `nm`, `objcopy`, and `objdump`), GNU Coreutils 8.30 (`dd`), and Perl 5.30.0. The project code distribution has been tested with these tools and works fine; any newer versions should also work, but may require some tweaking of either the source code or the `Makefile`.

The source code makes significant use of “GNU-isms”, so although you can try using a different compiler, it's quite possible you'll encounter strange compilation errors. Also, this has been tested on our Ubuntu® systems, and should work on any similar Linux® distribution (such as Debian®). However, it has not been tested extensively on macOS® systems, and has not been tested on Windows® systems.

For the debugging exercises, you will be using QEMU. The QEMU we are using is a patched version of QEMU 2.3.0. If you wish to install this on your own system for experimentation, you can download a ZIP archive containing the source code. It uses the standard Linux configuration script and installation tools. A newer version should also work, but won't have some of the changes made in this patched version (most notably, an unpatched version may go into a “boot loop” if the code being executed causes a certain type of fault; the patched version catches this and pauses to allow you to examine the register and memory contents).

You will also find several things linked from the course *Resources* page to be helpful:

- A book about the xv6 operating system we are studying in class: xv6: a simple, Unix-like teaching operating system (PDF)
- The xv6 source code, in book form (PDF) or as a collection of files.
- QEMU 2.3.0 User Documentation: HTML, PDF
- The code for the BIOS used in our version of QEMU, SeaBios 1.8.1 (also available for download as `seabios-rel-1.8.1.tar.gz`).

3. Assignment Details

3.1. 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.

3.1.1. Software Setup

The files you will need for this assignment can be retrieved on the CS systems using the following command:

```
$ get csci352 project1
```

This will unpack an archive file into a subdirectory named `project1/`. If you prefer, you can use this command to retrieve the distribution in an archive file:

```
$ get csci352 project1.tar
```

This will retrieve a `tar` file containing the source distribution.

In either case, the source code you retrieve is the initial skeleton of an operating system named JOS that bears some similarity to the one we have been studying in class. The `project1/` directory contains the following things:

```
$ ls -F project1
CODING  MKV*      boot/  fs/          inc/   lib/         user/
MK*     Makefile  conf/  gdbinit.tmpl kern/  mergedep.pl
```

The subdirectories contain source code for various parts of the system (including some you won't be using for this assignment). The files contain things we'll be looking at or using shortly.

3.1.2. x86 Assembly

If you are not already familiar with x86 assembly language, you will quickly become familiar with it. See my `x86-2-asm` lecture notes for much of the information you'll need.

If you prefer a “textbook”, one place to start is Paul Carter's PC Assembly Language Book *Note,* however, that the examples in the book are written for the NASM assembler, but we are 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. Fortunately, the conversion between the two is pretty simple, and is covered in Brennan Underwood's Brennan's Guide to Inline Assembly. (A local copy is also available.) In particular, the section entitled “The Syntax” in Brennan's Guide is a good (and brief) description of AT&T syntax.

The definitive reference for x86 assembly language programming is, of course, Intel's instruction set architecture reference, which you can find in many places online. See the *x86 Assembly Language Resources* section of my CSCI-452 Resources web page for links to many of these. One shorter and easier-to-navigate version of the Intel manuals is the 80386 Programmer's Reference Manual (PDF), found in my hardware documentation directory. This is a much older version of the Intel manuals, specifically for the 80386 processor; although it's not the most current, it describes all of the x86 CPU features you'll be using.

3.1.3. Simulating the x86

Instead of developing the operating system on a real, physical computer, we will use a program that faithfully (YMMV[1]) emulates a complete PC. (The code you write for the emulator will boot on a real PC, too.) Using an emulator simplifies debugging; you can, for example, set breakpoints inside of the emulated x86, which is difficult to do with the silicon version of an x86.

We will use a version of 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 assignment to step through the early boot process. We're using an older version of QEMU (2.3.0) that has been modified slightly to make it easier to deal with kernels that blow up.

3.1.4. Preparing the Bootable Image

To get started, extract the assignment files into your own directory as described above, then run `make V=0` to build the minimal boot loader and kernel you will start with. (It's a little generous to call the code we're running here a “kernel”, but it's close enough.) You should see something like the following:

```
$ cd project1
$ make V=0
+ 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
ld: warning: section `.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 412 bytes (max 510)
+ mk obj/kern/kernel.img
```

This compiles both the bootstrap and the kernel, leaving all the object and image files in the `obj/` subdirectory. The `V=0` argument causes `make` to print the brief summary lines you see above instead of the actual commands being executed; leave off that option for the full command trace.

Alternatives: There are also two shellscripts in the distribution, `MK` and `MKV`, which will run `make` and save the results into a file named `LOG` Running `./MK` will produce output that looks something like

```
$ ./MK
+ make > LOG 2>&1 ... done
Start: Tue May 27 15:40:57 EDT 2025
End:   Tue May 27 15:40:59 EDT 2025
```

and will leave the list of actual commands executed in `LOG`. Using `./MKV` script will save the command list in the log file, but will also show the commands in your shell window as they are executed. (In fact, these two scripts are actually one script that does different things based on the name used to invoke it.)

The `Makefile` puts all the compilation results into the `obj/` directory; after a successful compilation, it will contain the following:

```
$ ls -F obj
.deps  .vars.INIT_CFLAGS  .vars.KERN_CFLAGS  .vars.KERN_LDFLAGS  boot/  kern/
```

The files (`.deps`, etc.) are created by commands in the `Makefile` that help `make` figure out when things must be recompiled. Of more interest are the contents of the two subdirectories, whose names come from the source directories in the distribution:

```
$ ls -F obj/boot
boot*  boot.asm  boot.d  boot.o  boot.out*  main.d  main.o

$ ls -F obj/kern
console.d  entrypgdir.d  kdebug.d    kernel.img  printf.d    readline.d
console.o  entrypgdir.o  kdebug.o    kernel.sym  printf.o    readline.o
entry.d    init.d        kernel*     monitor.d   printfmt.d  string.d
entry.o    init.o        kernel.asm  monitor.o   printfmt.o  string.o
```

Some of these contain obvious things, but others may be new to you:

File(s)	Contents
`obj/*/*.d`	"Dependency" files produced by compiling the corresponding source file. These contain `make`-style dependency specifications; on subsequent compilations, these are concatenated and put into the `obj/.deps` file, which the `Makefile` includes automatically.
`*obj/*/.o`	Object files produced by compiling the corresponding source file.
`*obj/*/.asm`	Assembly language versions of some compiled files, produced to show you the actual assembly and machine-language code that will be executed along with their locations in the booted kernel.
`obj/boot/boot.out`	The compiled and linked bootstrap program. (This is a standard ELF format executable, which isn't usable as the boot sector in this format.)
`obj/boot/boot`	The "pure binary" boot sector created from the `boot.out` file. This is the code and data from that file, without any of the accompanying ELF header information, the symbol table, etc.
`kern/kernel`	The kernel, as an ELF executable.
`kern/kernel.asm`	A disassembled version of the kernel
`kern/kernel.img`	The bootable system image, consisting of the boot sector (from `boot`) followed by the pure binary version of the file `kernel`.
`kern/kernel.sym`	The symbol table from the `kernel` executable.

3.2. Part 2: Booting the System

Now you're ready to run QEMU on the bootable kernel image. To simplify things, the `Makefile` has a set of targets for running QEMU that provide all the options you will need to give it. The kernel image file is a "virtual hard disk" from QEMU's point of view, and contains both the boot loader and the kernel. The command `make qemu` executes QEMU with the options required to set the hard disk and direct serial port output to the terminal, and creates a virtual system console window. You'll see this in the shell window:

```
$ make qemu
... some echo commands ...
sed "s/localhost:1234/localhost:27111/" < gdbinit.tmpl > .gdbinit
/home/course/csci352/bin/qemu-system-i386 -drive file=obj/kern/kernel.img,index=
0,media=disk,format=raw -serial mon:stdio -gdb tcp::27111 -D qemu.log
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>
```

Each execution of `make` will determine whether or not any of the source files need to be recompiled before doing anything; the `echo` commands are related to setting up the `.deps` and `.vars.*` files in the `obj/` directory.

The first time you `make qemu`, the `Makefile` runs a `sed` command to set up a GDB configuration file in the project directory. It does this by editing a template (`gdbinit.tmpl` to insert a "random" network port number for GDB to use to communicate with the running QEMU process; we'll see how this works later. Once this file has been created, you won't see this command again (unless you use `make clean` to clean out the project directory).

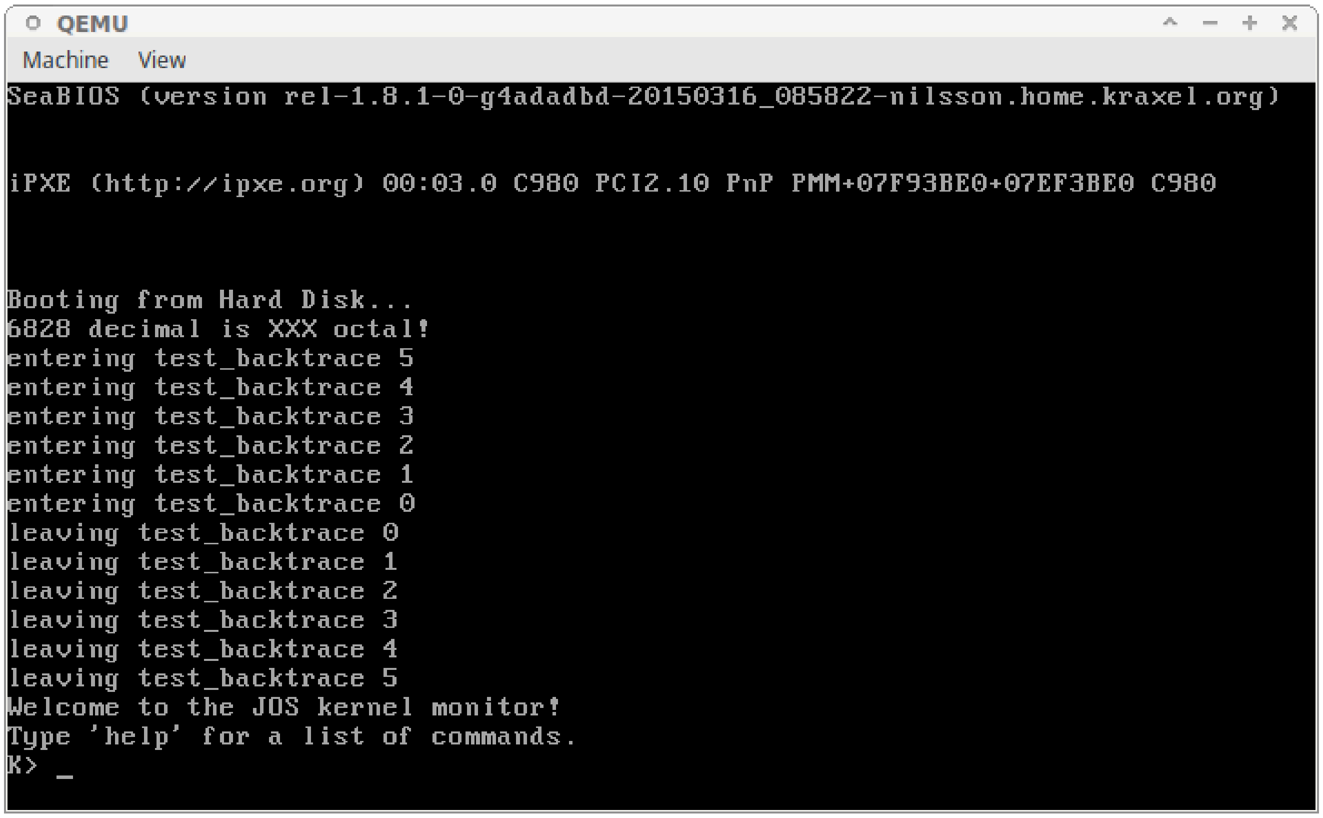
This will be followed by the command that actually starts QEMU; it's pretty long, so here it is broken into two pieces:

```
/home/course/csci352/bin/qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw
   -serial mon:stdio -gdb tcp::27111 -D qemu.log
```

This runs QEMU (located in the course account's `bin/` directory) with options that specify the disk to boot from (the file `obj/kern/kernel.img`), how to configure the virtual serial port, the network port to listen to if you're using GDB, and where to send log message output.

The line "`Booting from Hard Disk...`" is printed by QEMU as it starts up, locates the bootstrap, and begins the boot process; everything after that was printed by the skeletal kernel. (We'll come back to this.)

Along with this, a virtual system console window will pop up, looking something like Figure 1:



```
SeaBIOS (version rel-1.8.1-0-g4adadbd-20150316_085822-nilsson.home.kraxel.org)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F93BE0+07EF3BE0 C980



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> _
```

Figure 1: The QEMU Virtual Console Window

Back to the kernel output. You'll note that this output is repeated in the console window. The x86 system QEMU is simulating has a virtual *VGA graphics* I/O device, which is what is represented by the console window. It also has a *serial* I/O device, which we have told QEMU (via command-line arguments) to connect to the shell window. The simple kernel you're running is set up to replicate all console output to the serial output. Likewise, the 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.

If you prefer to run QEMU without the console window (or if you *must* do so because you don't have an X11 server running), you can run QEMU without the console window with the target `qemu-nox`. This adds a command-line argument to QEMU that stops it from creating the virtual console. Because all console output is replicated on teh serial output, you'll still see any messages printed to the console.

The kernel runs some internal tests, and then starts a small *monitor* (interactive control program). The monitor program prints the `K>` prompt; at that point, you can enter commands.

Note: If you need to quit QEMU, type `Ctrl-A` then `x` in the shell window where you ran QEMU; this tells QEMU to exit. Alternatively, you can type `Ctrl-A` then `c` in the shell window to switch to QEMU's interactive console and use its `q` command, or click the "close window" button on the console display window, or (if you are using the patched version of QEMU) select the "Quit" entry from the "Machine" menu button on the display window.

Important note: If the booted program should crash, the QEMU virtual console window may become unresponsive. In this situation, you should use `Ctrl-A` `c` in the shell window to switch to the QEMU interactive console; here you can use QEMU commands to poke around in the simulated system to try to figure out what went wrong. From there, use the `q` command to exit from QEMU. If things *really* went bad, or if QEMU itself crashes or exits strangely, you may find that your shell window is no longer echoing the characters you type. Use the shell command `stty sane` to return the shell window to a "sane" state.

There are only two commands you can give to the kernel monitor, `help` and `kerninfo`. You'll see output that looks something like this (although there may be some variation in some of the hex values):

```
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  f0101809 (virt)  00101809 (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.

3.2.1. The PC's Physical Address Space

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

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

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

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

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

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

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.

3.2.2. The ROM BIOS

In this portion of the assignment, you'll use QEMU's debugging facilities to investigate how an IA-32 compatible computer boots.

Open **two** terminal windows and `cd` both shells into your assignment directory. In one, enter `make qemu-gdb` (or `make qemu-nox-gdb`). This starts up QEMU, but tells QEMU to stop just before the processor executes the first instruction and to wait for a debugging connection from GDB. You should see something like this:

```
$ make qemu-gdb
... some echo commands ...
***
*** Now run 'make gdb'.
***
/home/course/csci352/bin/qemu-system-i386 -drive file=obj/kern/kernel.img,index=
0,media=disk,format=raw -serial mon:stdio -gdb tcp::27111 -D qemu.log  -S
```

In the second terminal, from the same directory, run `make gdb`. You should see something like this:

```
$ make gdb
gdb -q -n -x .gdbinit
+ target remote localhost:27111
warning: No executable has been specified and target does not support
determining executable automatically.
Try using the "file" command.
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
```

```
on
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

The code distribution contains a `.gdbinit` file that sets up GDB to debug the 16-bit code used during early boot, and directs it to attach to the listening QEMU. If it doesn't work, you may have to add the line

```
set auto-load safe-path /
```

to the file `$HOME/.gdbinit` in your account to convince GDB to process the `.gdbinit` provided with this assignment. GDB will tell you if you have to do this.

The 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 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 20-bit 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 + 0x0fff0    # easy--just append a 0.
   = 0xffff0
```

`0xffff0` is only 16 bytes before the end of the BIOS (`0x100000`), so we shouldn't be surprised that the first thing that the BIOS does is `jmp` backwards to an earlier location in the BIOS.

Question 1:

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 Storr's I/O Ports Description. No need to figure out all the details - just the general idea of what the BIOS is doing first. Describe (briefly) what you have determined is going on.

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.

3.3. Part 3: 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*, because 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 address `0x7c00` (called the *load address*), 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 our work, however, we will use the conventional hard drive boot mechanism, which means that our boot loader must fit into a whopping 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:

- 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 Carter's *PC Assembly Language* book, 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.

- 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. (While we will discuss device communication to some extent, it will not be a major focus in this course, and you will not do any device I/O programming for this assignment.)

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 `Makefile` creates *after* compiling the boot loader. This disassembly file makes it easy to see exactly where in physical memory all of the boot loader's code resides, and makes it easier to track what's happening while stepping through the boot loader in GDB. Likewise, `obj/kern/kernel.asm` contains a disassembly of the kernel, which can often be useful for debugging.

In addition to setting breakpoints at labels (e.g., function entry points), GDB allows you to set them at arbitrary memory addresses. For example, `b *0x7c00` (note the "`*`") 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.

The MIT 6.828 course lab tools guide web page has a nice section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Internally, GDB operates with a “target architecture” as its model for how the hardware should work. When you started GDB in your second shell window, one of the lines printed was

```
    The target architecture is assumed to be i8086
```

You can verify this with the `show arch` command, or change it with the `set arch a`, where *a* is the name of the target architecture you want to use. If you find that GDB is printing machine instructions strangely, you might try setting the architecture to `i386` or to `auto`.

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 `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 disassembly in `obj/boot/boot.asm` and 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.

Question 2:

A. At what point does the processor *first* start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

B. What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?

C. *Where* is the first instruction of the kernel?

D. 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?

3.3.1. Loading the Kernel

We will now look in further detail at the C language portion of the boot loader, in `boot/main.c`.

Before doing so, this may be a good time to stop and review some of the basics of C programming. Read about programming with pointers in C. The classic reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'), but the CSCI-452 Resources web page has links to several others.

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 printed lines 1 and 6 come from, how all the values in printed 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 (e.g., a tutorial by Ted Jensen that cites K&R heavily), though not as strongly recommended.

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 the future, 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 have some idea of what an ELF binary is.

When you compile and link a C program such as the 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, 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 purposes of our study, 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 examine the full list of the names, sizes, and link addresses of all the sections in the kernel executable by typing:

```
$ objdump -h obj/kern/kernel
```

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 here.)

Typically, the link and load addresses are the same. For example, look at the `.text` section of the boot loader:

```
$ objdump -h obj/boot/boot.out
```

The boot loader uses the ELF *program headers* to decide how to load the sections. he 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:

```
$ objdump -x obj/kern/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 `boot/main.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 `boot/Makefrag`, so the linker will produce the correct memory addresses in the generated code.

Question 3:

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` again afterward! Describe what you changed and what the effect was; was the effect what you expected, or something different?

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:

```
$ objdump -f obj/kern/kernel
```

You should now be able to understand the minimal ELF loader in `boot/main.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.

Question 4:

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.)

3.4. Part 4: The Kernel

We will now start to examine the minimal 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.

3.4.1. 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.

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.

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).

Question 5:

Use QEMU and GDB to trace into the 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 `kern/entry.S`, trace into it, and see if you were right.

3.4.2. 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 1:

In the `cprintf()` implementation, a small fragment of code has been omitted - the code necessary to print numbers in octal using patterns of the form "%o". Find and fill in this code fragment.

Question 6:

A. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?

B. 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);
```

C. 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?

3.4.3. The Stack

In the final exercise of this assignment, 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.

Question 7:

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.

Question 8:

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 CS systems or from the link given earlier. 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 name `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:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
        ...
```

Each line contains an `ebp`, `eip`, and `args`. 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?)

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.

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 2:

Implement the backtrace function as specified above. Use the same format as in the example.

If you use `read_ebp()`, note that GCC may generate "optimized" code that calls `read_ebp()` *before* `mon_backtrace()`'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of `mon_backtrace()` and make sure the call to `read_ebp()` is happening after the function prologue.

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`.

4. Turning In Your Solution

Submit your answers to the questions in the previous section on the CS systems with the command

```
$ try wrc-grd is-1 answers.txt  files
```

where `answers.txt` contains your answers to the 8 questions in the sections above, and *`files`* are the names of the files containing your code solutions for the 2 exercises above.

Your `answers.txt` file should be a plain-text file (i.e., no PDF, MSWord, etc. files!). You may also submit a plain-text file named `README` if you wish to send additional comments or information.

If you are working as a team, each team should submit only *one copy* of its answers. All members of the team will receive the same grade for this assignment.

5. Future Work

For those who are interested in other possible modifications to the simple kernel, here are some “challenge” problems to think about. (These are not to be submitted as part of this assignment; the idea is to get you thinking about other ideas for enhancements and modifications to the basic system.)

Challenge 1:

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. You can find information about programming the VGA display on many websites; one such page is MIT's 6.828 course reference page. 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.

Challenge 2:

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 `objdump -h obj/kern/kernel`
- run `objdump -G obj/kern/kernel`
- run `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:
        ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
                                 kern/monitor.c:143: monitor+106
        ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
                                 kern/init.c:49: i386_init+59
        ebp f010fff8  eip f010003d  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`).

Print the file and function names on a separate line, to make the results easier to read.

Tip: `printf()` format strings provide an easy, albeit obscure, way to print non-NUL-terminated strings like those in STABS tables. `printf("%.*s", length, string)` prints at most `length` characters of `string`. Take a look at the `printf()` man page to find out why this works.

You may find that 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 `Makefile`, the backtraces may make more sense (but your kernel will run more slowly).

---

[1] *Your mileage may vary. Void where prohibited by law. Standard disclaimers apply.*

Ubuntu® is a registered trademark of Canonical Ltd.