Note : if you got stuck in a part continue Reading , maybe a later part will explain it better or refer to the original document http://pdosnew.csail.mit.edu/6.828/2014/labs/lab1/

Note : Kernel == Mini OS == JOS kernel

Table of Content

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 6.828 kernel, which resides in the boot directory of the lab tree.
- Finally, the third part delves into the initial template for our 6.828 kernel itself, named JOS, which resides in the kernel directory.

Software Setup

- Assuming you are using ubuntu , run this command

```
objdump -i | grep 'elf32-i386'
```

the result shouldn't be empty, if it was empty please visit
http://pdos.csail.mit.edu/6.828/2014/tools.html

- Run the following commands in order to install tools (gcc , gdb , git , qemu) and get lab you can copy them directly into a script and run it , please take a look at the comments to understand what they do (don't be a copy paste machine)

```
sudo apt-get install gcc
#this command installs gcc compiler
sudo apt-get install gdb
#this command installs gdb debugger
sudo apt-get install git
#this command installs git source controllers (something like svn)
mkdir ~/OS_project
#this command, mmmm you should know what it does :D , this will be the directory we works on please remember it
cd ~/OS_project
git clone http://pdos.csail.mit.edu/6.828/2014-jos.git lab
#this command receives a copy from the lab code from the MIT repository
mkdir ~/qemu
#this command creates a folder to install in it our virtual machine program  which is called qemu
cd ~/qemu
git clone https://github.com/geofft/qemu.git -b 6.828-1.7.0
#this command retrieve qemu source code , inorder to build it and use it
./configure --disable-kvm [--prefix=PFX] [--target-list="i386-softmmu x86_64-softmmu"]
#this command setup some configuration to the qemu build
make && make install
#this command builds and installs your qemu , if failed check the note below
echo “export PATH=~/qemu/PFX/bin:\$PATH” >> ~/.bashrc
```

```
#this command add the path of your qemu to your PATH environment variable found in ~/.bashrc
```

- if you got an error in the make install run this command and re-run “make && make install”

```
sed -i ‘s#~/qemu/$(DESTDIR)#$(DESTDIR)#g’ Makefile*
```

For tools usage visit http://pdos.csail.mit.edu/6.828/2014/labguide.html

Hand-In Procedure

- Handed out Saturday, November 8, 2014
- Due Saturday, November 22, 2014
- Groups of : 2
- Delivery Mail Subject : [MiniOS][credit/semester] delivery#1
  - write either credit or semester not both :)
- Delivery Mail : d.tantawy@gmail.com
- Delivery Mail Content: your Names in arabic
- Deliverables : Put text file with your names , and a pdt containing the answer of the questions in the folder containing your project, compressed it and send .

Part 1: PC Bootstrap

Simulating the x86

1. First Build your Kernel (MiniOS)

```
cd  ~/OS_project/lab/
# to get into the code top directory
make
#it calls by default the makefile and run the all target which build the bootloader and the kernel
the output of the command as the following
+ as kern/entry.S
+ cc kern/init.c
+ cc kern/console.c
```

```
+ cc kern/monitor.c
+ cc kern/printf.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 414 bytes (max 510)
+ mk obj/kern/kernel.img
```

This steps built your MiniOS at obj/kern/kernel.img.

2. Load your QEMU with the OS image you just created

you will supply the image as emulated PC's "virtual hard disk." This hard disk image contains both our boot loader (obj/boot/boot) and our kernel (obj/kernel).

```
make qemu
#this command starts your qemu and load it with the image
#as you previous know that make checks the makefile and applies the commands at the target specified
 #make qemu-nox
#this command starts your qemu in non-graphical mode (the kernel will run in the terminal) and load it with the image
#make qemu-gdb
#like make qemu but it opens a port for gdb to allow it to debug the kernel
#make qemu-nox-gdb
#like make qemu-nox but it opens a port for gdb to allow it to debug the kernel
#run only one command as you need it

#the following text will appear in both qemu window and terminal unless you are in non-graphical mode it will appears only on the terminal

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 serial port, which QEMU in turn outputs to its own standard output. 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.

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:
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101a75 (virt)  00101a75 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112960 (virt)  00112960 (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 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,

```
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)
```

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.)**

Part 2: The Boot Loader

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 (at least get a general idea). The boot loader must perform two main functions:

1. First, the boot loader switches the processor from real mode to *32-bit protected mode*.

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.

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. Likewise, obj/kern/kernel.asm contains a disassembly of the JOS kernel, which can often be useful for debugging.

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/*N*i *ADDR***, where *N* is the number of consecutive instructions to disassemble and *ADDR* is the memory address at which to start disassembling.

Requirement 1

Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

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.

Answer the following :

- 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

(how linking happens, nice theoretical part)

We will now look in further detail at the C language portion of the boot loader, in boot/main.c. 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 asobj/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. The Wikipedia page has a short description.

For purposes of our project, 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 ininc/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:

```
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)
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. 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:

```
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.

Requirement 2

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!

Answer
what was the instruction that breaks

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.

Requirement 3

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*/N*x *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).

Answer
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 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.

(you can skip this part for now , though it is recommended to read it)

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 ofkern/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 inkern/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).
Exercise (not required). 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, %cr0in 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.

Requirement 4

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.

Answer the following

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     }
```

3. 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 at 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?
7. 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.

Requirement 5

Answer

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.

Requirement 6

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?

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 inkern/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
  ...
```

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?)
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.

Requirement 7 (implementation)

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.
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.

Requirement 8 (implementation continue)

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:
  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).

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

Tip: printf format strings provide an easy, albeit obscure, way to print non-null-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 GNUMakefile, the backtraces may make more sense (but your kernel will run more slowly)