Os lab1: Booting a PC

陈炜栋 11300240057

November 23, 2013

Contents

1 PC Bootstrap

1.1 Getting Started with x86 assembly

> Exercise 1: Familiarize yourself with the assembly language materials available on the 6.828 reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT\&T assembly syntax we'll be using with the GNU assembler in JOS.

花了时间阅读。内容比较多，只能用的时候再回来查。

1.2 Simulating the x86

好奇kernel.img里是什么东西。有条命令 **xxd - make a hex dump or do the reverse** 可以将二进制文件以十六进制的形式呈现。

xxd kernel.img

```
1 0000000: fafc 31c0 8ed8 8ec0 8ed0 e464 a802 75fa
2 0000010: b0d1 e664 e464 a802 75fa b0df e660 0f01
3 0000020: 1664 7c0f 20c0 6683 c801 0f22 c0ea 327c
4 0000030: 0800 66b8 1000 8ed8 8ec0 8ee0 8ee8 8ed0
5 0000040: bc00 7c00 00e8 c100 0000 ebfe 0000 0000
6 0000050: 0000 0000 ffff 0000 009a cf00 ffff 0000
7 0000060: 0092 cf00 1700 4c7c 0000 9090 5589 e5ba
8 0000070: f701 0000 ec25 c000 0000 83f8 4075 f5c9
9 0000080: c355 89e5 578b 7d0c e8df ffff ffba f201
10 0000090: 0000 b001 eeb2 f389 f8ee 89f8 c1e8 08b2
11 00000a0: f4ee 89f8 c1e8 10b2 f5ee c1ef 1889 f883
12 00000b0: c8e0 b2f6 eeb2 f7b0 20ee e8ad ffff ff8b
```

- 后面还有一列输出这些十六进制解释出来的字符串。这里不关心，忽略掉。
- 再观察 xxd boot 的二进制文件，及xxd kernel,其中boot占用512bytes,Kernel与从Kernel.img512+开始相同。看到后面就会知道，其实kernel.img正是将boot放在第一个sector，紧接将Kernel放在第二个sector.

vim kern/Makefrag

```
75 # How to build the kernel disk image
76 $(OBJDIR)/kern/kernel.img: $(OBJDIR)/kern/kernel $(OBJDIR)/boot/boot
77         @echo + mk $@
78         $(V)dd if=/dev/zero of=$(OBJDIR)/kern/kernel.img~ count=10000 2>/dev/null
79         $(V)dd if=$(OBJDIR)/boot/boot of=$(OBJDIR)/kern/kernel.img~ conv=notrunc 2>/dev/null
80         $(V)dd if=$(OBJDIR)/kern/kernel of=$(OBJDIR)/kern/kernel.img~ seek=1 conv=notrunc
               2>/dev/null
81         $(V)mv $(OBJDIR)/kern/kernel.img~ $(OBJDIR)/kern/kernel.img
```

- 从上面的编译来看， line 78 为整个文件占了10000个sector，一共5120000=0x4E2000 byte,下图是kernel.img二进制文件的最后三行，发现对78行的猜测没有错。
- line 79 , line 80 依次将boot,kernel 复制到kernel.img。 其中boot占据第一个sector,seek=1表示kernel从第二个sector开始复制

xxd kernel.img

```
319998 04e1fd0: 0000 0000 0000 0000 0000 0000 0000 0000
319999 04e1fe0: 0000 0000 0000 0000 0000 0000 0000 0000
320000 04e1ff0: 0000 0000 0000 0000 0000 0000 0000 0000
```

1.3 The PC's Physical Address Space

- Kernel.img ELF 头文件被放在0x10000,4KB， 搞不清楚什么时候在哪个环节被放入的？似乎bootmain()里用的时候似乎理所应当地就知道ELF头文件就在0x10000?
- BIOS 为 1MB的最上的64KB，程序第一步进去在[0xf000:0xfff0],就是在BIOS，BIOS应该是在JOS之外的，这个入口应该是约定俗成的，BIOS是制造厂商设置的

1.4 The ROM BIOS

BIOS 找到一个可启动的介质，将第一个扇区读入到0x7c00到0x7dff,然后jmp到该地址，将控制权交给boot loader
在阅读代码的遇到一些GAS命令：

- CLI: Clear the interrupt flag. 昨晚lab3再来做lab1才知道是什么意思..， 机器无法产生中断，但是程序可以产生软件中断，来改变代码执行流程，NMI中断不能被阻止。
- CLD: clear the direction flag,Registers will increment reading forward. 这个应该是指eip吧。
- in,out指令，从或者向指定的端口读写数据。

Listing:

```
Exercise 2: Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few
    more instructions, and try to guess what it might be doing. You might want to look at
    Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference
    materials page. No need to figure out all the details - just the general idea of what the
    BIOS is doing first.
```

手册这么长。。。

2 The Boot Loader

Listing:

```
Exercise 3: 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.

Be able to answer the following questions:

- At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?
- Where is the first instruction of the kernel?
- How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

1. 处理器从BIOS进入boot loader后，在boot/boot.S中，boot loader 将寄存器cr0 的末位更改为1，是的处理器从实模式改到保护模式。

boot/boot.S

```
48    lgdt    gdtdesc
49    movl    %cr0, %eax
50    orl     $CR0_PE_ON, %eax
51    movl    %eax, %cr0
```

2. boot loader执行的最后一条指令为将内核ELF文件载入内存后，调用的entry-point,在boot/main.c中的第58行

boot/main.c

```
56 // call the entry point from the ELF header
57 // note: does not return!
58 ((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))();
```

3. kernel.ld中的入口地址为0xf010000c,但是为什么最后载入到了0x10000c,张弛的报告说在boot/main.c手动做过一次转化。但我仔细找了boot/main.c也没有找到，bootmain只是按照ELF定义的p_pa载入，所以我理解的就是虽然kernel.ld中定义的入口地址是0xf010000c,但是最后解释成物理地址就是0x10000c,而不是在bootmain里进行转化

   Anyway,在0x10000c中设置断点之后,得出kernel的第一条指令是 0x10000c: movw $0x1234,0x472

console:

```
The target architecture is assumed to be i8086
[f000:fff0]     0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x10000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472
Breakpoint 1, 0x0010000c in ?? ()
(gdb)
```

4. boot loader 从ELF文件的文件头可以知道该ELF文件被分成了多少section和多少program

2.1 Loading the Kernel

Listing:

```
Exercise 4. Read about programming with pointers in C. The best reference for the C language
    is The C Programming Language by Brian Kernighan and Dennis Ritchie (known as 'K\&R').
    We recommend that students purchase this book (here is an Amazon Link) or find one of MIT
    's 7 copies.
Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K\&R.
    Then download the code for pointers.c, run it, and make sure you understand where all of
    the printed values come from. In particular, make sure you understand where the pointer
    addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there,
    and why the values printed in line 5 are seemingly corrupted.
There are other references on pointers in C, though not as strongly recommended. A tutorial
    by Ted Jensen that cites K\&R heavily is available in the course readings.
```

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

问题不大，强制转化成数组指针，运算后在强制转化成整数指针的需要稍微注意下小端法，对号入座。

Listing:

Exercise 5: 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!

Link address 就是 line 27 的 0x7c00 这个不能乱改。。乱改读入的地址不对，就BIOS就找不到Boot loader了。

boot/Makefrag:

```
25 $(OBJDIR)/boot/boot: $(BOOT_OBJS)
26         @echo + ld boot/boot
27         $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
28         $(V)$(OBJDUMP) -S $@.out >$@.asm
29         $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
30         $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

Listing:

Exercise 6: We can examine memory using GDB's x command. The GDB manual has full details, but for now, it is enough to know that the command x/Nx ADDR prints N words of memory at ADDR. (Note that both 'x's in the command are lowercase.) Warning: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

看结果：

console:

```
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0:    ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] 0x7c00:     cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b 0x10000c
Function "0x10000c" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
0x10000c:       movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010

}
```

3 Part 3: The Kernel

3.1 Using virtual memory to work around position dependence

Listing:

```
Exercise 7: Use QEMU and GDB to trace into the JOS kernel and stop at the movl %eax, %cr0.
    Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction
    using the stepi GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make
    sure you understand what just happened.
What is the first instruction after the new mapping is established that would fail to work
    properly if the mapping weren't in place? Comment out the movl \%eax, \%cr0 in kern/entry
    .S, trace into it, and see if you were right.
```

按照提示一步一步做下来会得到下图。

先b *0x10000c 不能是0xf010000c 因为gdt还没有载入= =

下图表明在经过 mmov %eax,%cr0 之后，0xf010000c 被映射到0x10000c

console:

```
0x100025:       mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x6000b812      0x220f0011      0xc0200fd8
(gdb) x/8x 0xf010000c
0xf010000c <entry>:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xf010001c <entry+16>:  0xffffffff      0xffffffff      0xffffffff      0xffffffff
(gdb) si
0x100028:       mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/8x 0xf010000c
0xf010000c <entry>:     0x7205c766      0x34000004      0x6000b812      0x220f0011
0xf010001c <entry+16>:  0xc0200fd8      0x0100010d      0xc0220f80      0x10002fb8
(gdb)
}
```

line 82 如果注释掉，那么line 87 $relocated = 0xf010002c ,这时 line 89 jmp的时候会出问题，因为这个时候，实际的代码位置在0x10002c没有经过一次减KERNELBASE 的运算，那么会导致找到莫名奇妙的位置去。

导致QEMU Triple fault.

vim kern/entry.S

```
75 # Load the physical address of entry_pgdir into cr3.  entry_pgdir
76 # is defined in entrypgdir.c.
77 movl    $(RELOC(entry_pgdir)), %eax
78 movl    %eax, %cr3
79 # Turn on paging.
80 movl    %cr0, %eax
81 orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
82 movl    %eax, %cr0
83
84 # Now paging is enabled, but we're still running at a low EIP
85 # (why is this okay?).  Jump up above KERNBASE before entering
86 # C code.
87 mov     $relocated, %eax
88 jmp     *%eax
```

3.2 Formatted Printing to the Console

Listing:

```
Exercise 8: We have omitted a small fragment of code - the code necessary to print octal
   numbers using patterns of the form "%o". Find and fill in this code fragment.}
```

vim lib/printfmt

```
207 case 'o':
208         num = getint(&ap, lflag);
209                 if ((long long) num < 0) {
210                         putch('-', putdat);
211                         num = -(long long) num;
212                 }
213         base = 8;
214         goto number;
```

Be able to answer the following questions:

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

   console.c主要提供一些与硬件直接进行交互的接口以便其他车工女婿进行输入输出的调用。

其中与printf.c进行交互的主要是cputchar函数

2. Explain the following from console.c:

vim lib/printfmt

```
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
}
```

上面这段代码主要用在打印后检测是否满屏，如果满屏，则将最后一行空出来，全部置为空格。

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

   int x = 1, y = 3, z = 4; cprintf("x %d, y %x, z %d", 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.
   fmt指的是格式字符串，ap指的是不定参数表的第一个参数的地址。

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.
   打印出的是He110 World! 可以把这段代码嵌入到monitor.c里面运行。

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);
   va_arg从ap指针不断往后去得到。如果给的参数数量不足实际要打印的数量，那么ap就调到一个位置内存区域，打印出一个未知的东西。

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?

具体变长参数依赖 于inc/stdarg.h中va_arg的实现

3.3 The Stack

Listing:

```
Exercise 9: 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?
```

下列 文件line 69 初始化栈， line 91 .space KSTKSIZE 为栈保留了8*4KB的空间。栈从上往下，栈的增长从地址高向地址低的地方。

vim kern/entry.S

```
69  relocated:
70
71          # Clear the frame pointer register (EBP)
72          # so that once we get into debugging C code,
73          # stack backtraces will be terminated properly.
74          movl    $0x0,%ebp                       # nuke frame pointer
75
76          # Set the stack pointer
77          movl    $(bootstacktop),%esp
78
79          # now to C code
80          call    i386_init
81
82          # Should never get here, but in case we do, just spin.
83  spin:   jmp     spin
84
85
86  .data
87
88  .p2align        PGSHIFT         # force page alignment
89          .globl          bootstack
90  bootstack:
91          .space          KSTKSIZE
```

```
92          .globl          bootstacktop
93  bootstacktop:
```

Listing:

```
Exercise 10: To become familiar with the C calling conventions on the x86, find the address
    of the test\_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and
    examine what happens each time it gets called after the kernel starts. How many 32-bit
    words does each recursive nesting level of test\_backtrace push on the stack, and what
    are those words?
Note that, for this exercise to work properly, you should be using the patched version of
    QEMU available on the tools page or on Athena. Otherwise, you'll have to manually
    translate all breakpoint and memory addresses to linear addresses.
```

line 77 按照惯例push ebp +4

line 79 push +4

line 80 sub +12

line 83 push +4

iine 84 push +4

line 85 call +4

line 87 add -16

line 91 sub +12

line 93 push +4

line 94 call +4

总共32byte

也就是说对于每一层test_backtrace需要用掉32个byte

vim obj/kern/kernel.asm

```
74  void
75  test_backtrace(int x)
76  {
77  f0100040:       55                      push   %ebp
78  f0100041:       89 e5                   mov    %esp,%ebp
79  f0100043:       53                      push   %ebx
80  f0100044:       83 ec 0c                sub    $0xc,%esp
```

```
f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
        cprintf("entering test_backtrace %d\n", x);
f010004a:       53                      push   %ebx
f010004b:       68 e0 18 10 f0          push   $0xf01018e0
f0100050:       e8 f4 08 00 00          call   f0100949 <cprintf>
        if (x > 0)
f0100055:       83 c4 10                add    $0x10,%esp
f0100058:       85 db                   test   %ebx,%ebx
f010005a:       7e 11                   jle    f010006d <test_backtrace+0x2d>
                test_backtrace(x-1);
f010005c:       83 ec 0c                sub    $0xc,%esp
f010005f:       8d 43 ff                lea    -0x1(%ebx),%eax
f0100062:       50                      push   %eax
f0100063:       e8 d8 ff ff ff          call   f0100040 <test_backtrace>
```

Listing:

```
Exercise 11: 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.
```

和下一题一起做。

Listing:

```
Exercise 12: 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 inlines 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).

kern/kdebug.c里面主要是阅读一下stab_binseach函数，计算行数的那里，只需模仿同一个函数中计算file及fun的。再参考注释不难弄出来。

下列代码，是添加在kern/monitor.c中的代码。

栈里面的结构是

- ArgN,ArgN-1,...Arg0
- eip
- last ebp

这样就不难得出线面的代码，cprintf("%.*s") 可以代入两个参数输出变长的字符串。

vim kern/monitor.c

```
59 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
60 {
61         // Your code here.
62     uint32_t *ebp, *eip;
63     uint32_t arg0,arg1,arg2,arg3,arg4;
64
65     ebp = (uint32_t *) read_ebp();
66     eip = (uint32_t *) ebp[1];
67     arg0 = ebp[2];
68     arg1 = ebp[3];
69     arg2 = ebp[4];
70     arg3 = ebp[5];
71     arg4 = ebp[6];
72
73     cprintf("Stack backtrace: \n");
74     struct Eipdebuginfo info;
75     while (ebp != 0){
76         cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n",ebp,eip,arg0,arg1,
              arg2,arg3,arg4);
77         debuginfo_eip((uintptr_t)eip,&info);
78         cprintf("         %s:%d: %.*s",info.eip_file,info.eip_line,info.eip_fn_namelen,
79                 info.eip_fn_name);
80
81         cprintf("+%d\n",(uint32_t) eip-info.eip_fn_addr);
82         ebp = (uint32_t *) ebp[0];
```

```
83          eip = (uint32_t *) ebp[1];
84          arg0 = ebp[2];
85          arg1 = ebp[3];
86          arg2 = ebp[4];
87          arg3 = ebp[5];
88          arg4 = ebp[6];
89      }
90
91
92      return 0;
93  }
```

make grade 的结果如下。

console:

```
running JOS: (0.7s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

4 补充题

请回答一下两个question，并附在报告尾部(10%)。

(1)同学们在做Part 2: The Boot Loader部分实验的时候会阅读到以下代码：

Listing:

```
# Bootstrap GDT

.p2align 2                                                    # force 4 byte
    alignment

gdt:
```

```
  SEG_NULL                                                    # null seg

  SEG(STA_X|STA_R, 0x0, 0xffffffff)                  # code seg

  SEG(STA_W, 0x0, 0xffffffff)                          # data seg

gdtdesc:

  .word   0x17                                                # sizeof(gdt) - 1

  .long   gdt                                                  # address gdt

(/boot/boot.S:75-84)
```

此段代码的作用是什么？为什么要引入GDT？(简要回答，100字以内) 在 32 位 protected Mode 下,一个段描述符寄存器是一个 64 位的。

如果直接通过 64-bit 的段描述符来引用一个段的时候,必须使用一个 64-bit 长的段寄存 器来装入这个段描述符,但是为了保持向后兼容,段寄存器仍旧规定为 16-bit,那么显然 无法用一个 16-bit 的寄存器来直接引用 64-bit。这个时候就引入了 GDT,将这些长度为 64bit 的段描述符放入一个表中,将段寄存器的下标索引用来间接引用。

GDT 就是一张表,引入的原因之一是为了向后兼容。

(2)同学们在做exercise11的时候会碰到以下代码：

Listing:

```
test_backtrace(int x)

{

f0100040:        55                          push   %ebp

f0100041:        89 e5                      mov    %esp,%ebp

f0100043:        53                          push   %ebx

f0100044:        83 ec 14                   sub    $0x14,%esp
```

请问为什么是sub $0x14,最后总的栈的大小应该是32的倍数。这个从Exercise 10 可以猜一下。出去eip和ebp等必须的。其他的应该是为了存储临时变量的。