

操作系统上机实验

Lab 1：启动计算机

实验报告

October 21, 2019

曹元议 1711425
黄艺璇 1711429
王雨奇 1711299
张瑞宁 1711302
阔坤柔 1611381
冯晓妍 1513408

Abstract

在操作系统课程学习的过程中，仅有理论学习是不够的，因此我们通过动手实验来加深对操作系统理论知识的理解。在本次实验中，我们通过观察 PC 的启动和引导过程、内存的地址空间、控制台的输入和输出、函数调用的过程和堆栈的变化来了解操作系统在启动过程中的步骤和流程，并且通过这个过程，熟悉了实验使用的 QEMU 平台和 JOS 操作系统内核，也为后续的实验打下了基础。

Keywords: 操作系统；QEMU；JOS；Boot Loader；内核；堆栈

Contents

1	小组分工情况	1
2	计算机的引导	1
2.1	练习 1	1
2.2	Simulating the x86	2
2.3	PC 的物理地址空间	3
2.4	BIOS	5
3	Boot Loader	5
3.1	练习 2	5
3.2	问题 1	6
3.3	载入内核	11
4	The Kernal	14
4.1	问题 2	14
4.2	作业 1	17
4.3	堆栈	19
4.4	练习 3	20
4.5	作业 2	22
4.6	挑战作业	25
5	总结	35

1 小组分工情况

	问题解决	报告整理	报告检查	小组讨论	Lab 1 任务
曹元议	√	√		√	问题 2、作业 2、挑战作业、练习
黄艺璇			√	√	
王雨奇			√	√	
张瑞宁			√	√	问题 1、作业 1
阔坤柔	√		√	√	
冯晓妍			√	√	

Table 1: 小组分工

组长的话：由于通知了 Lab 1 和 Lab 2 的作业分开交之前我们就已经分好任务了，因此未分配到 Lab 1 的组员都被分到了 Lab 2 的任务，但是大家都参与了小组讨论，我负责整理报告（当然，其他组员如果被分到任务也要写自己负责的这一部分的报告），除了我以外的其他组员都参与了报告的检查及修改工作。以后我分任务会注意的。

2 计算机的引导

2.1 练习 1

2.1.1 题目原文

参考书籍使用的 `NASM assembler`，而我们使用的是 `GNU assembler`，两者语法略有不同，可参考 `Brennan's Guide to Inline Assembly` 查阅相关不同之处。

2.1.2 题目大意

在汇编语言课程的学习中，我们学习的是 Intel 8086 汇编语言，NASM 汇编器使用的就是 Intel x86 汇编语言。而实验中接触的汇编语言是 AT&T 汇编语言，由 GNU 汇编器使用。两者虽然都是 x86 汇编语言，但语法上有一些差别。比较 Intel 8086 汇编语言和 AT&T 汇编语言在语法格式上的差异。

2.1.3 回答

分六个方面进行比较：

1. 使用寄存器: AT&T 必须有%前缀, 例如%eax; 而 Intel 没有前缀, 例如eax。
2. 源/目的操作数方向: AT&T 的源操作数在左侧, 目的操作数在右侧; Intel 正好相反。
3. 常数/立即数值格式: AT&T 必须有\$前缀, 十六进制数前面要加0x。例如movl \$_booga, %eax、movl \$0xd00d, %ebx; 而 Intel 没有前缀, 但是十六进制数有h后缀, 二进制立即数有b后缀, 例如mov eax, _booga、mov ebx, d00dh
4. 操作数的长度: AT&T 必须在指令助记符后面附加b、w、l分别表示字节 (byte, 8 bits)、字 (word, 16 bits) 和长字 (long, 32 bits) 例如: movl (%ebx), %eax; 而 Intel 可在操作数前面附加byte ptr、word ptr、dword ptr, 如果不加则取寄存器操作数的长度或取数据段定义的类型 (db、dw、dd), 例如: mov eax, dword ptr [ebx]。
5. 内存的寻址: AT&T 的格式为imm32(basepointer, indexpointer, indexscale)表示含义为imm32 + basepointer + indexpointer * indexscale, 例如: movl -4(%ebp), %eax, 实模式下的跨段寻址还需在前面加上%segreg:前缀: movb \$4, %fs:(%eax); Intel 的格式为[basepointer + indexpointer * indexscale + imm32], 表示含义同前, 例如: mov eax, [ebp-4], 实模式下的跨段寻址还需在前面加上segreg:前缀: mov fs:eax, 4。
6. 无条件跳转指令和调用指令: AT&T 如果是间接寻找跳转目标还需在操作数前加上*前缀, 例如: jmp/call *%eax, 远程跳转或调用需要在助记符前加上l前缀, 例如: ljmp/lcall \$section, \$offset; Intel 的格式较简单: jmp/call eax, 远程跳转或调用需要在助记符后面加上far, 例如: jmp/call far section:offset。

2.2 Simulating the x86

在本次实验以及后续的实验中, 我们需要使用 QEMU 模拟机来模拟一台真正的 PC。我们首先在宿主机的 VMware Workstation 中安装了 Ubuntu 虚拟机。再根据实验指导和实验文件, 在这个 Ubuntu 虚拟机中把所需的环境配置好之后, 就在终端输入命令make qemu来启动 QEMU, 加载 JOS 内核。

启动 QEMU 之后, JOS 的内核被加载, 操作系统在 QEMU 中正常地运行。控制台的输出如下:

```
6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
```

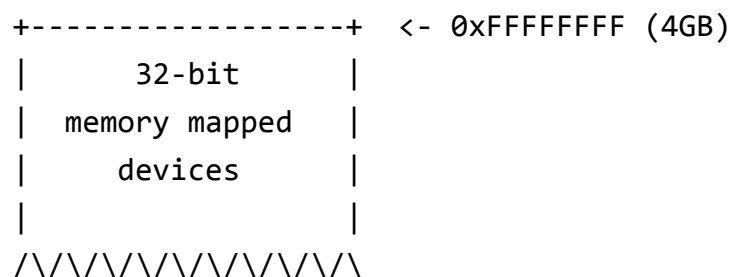
```
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

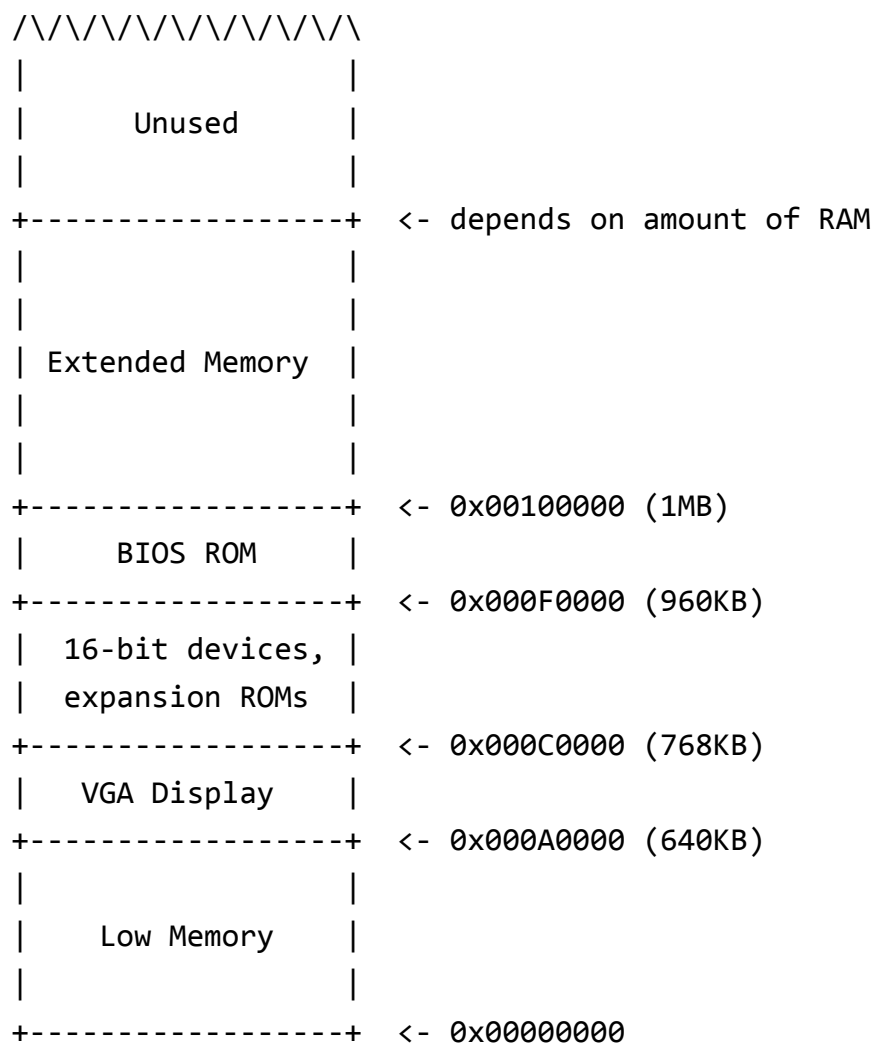
由此可见, 初始的 JOS 内核只支持两个命令: `help` 和 `kerninfo`。`help` 命令显示 JOS 所支持命令的用途, `kerninfo` 命令打印 JOS 的内核信息。JOS 的两条命令显示如下:

```
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   f0101a1f (virt) 00101a1f (phys)
    edata   f0112300 (virt) 00112300 (phys)
    end     f0112944 (virt) 00112944 (phys)
Kernel executable memory footprint: 75KB
K>
```

2.3 PC 的物理地址空间

一台 PC 的物理地址空间大致划分如下:





在早期 Intel 处理器（例如 8086）上，只有 16 位地址线，最多只能使用 1MB 的物理内存，即物理地址空间范围为 $0x00000000 \sim 0x000FFFFF$ 而不是地址 $0xFFFFFFFF$ 。最底下的 640KB 也是早期 PC 所唯一能使用的 RAM， $0x000A0000 \sim 0x000FFFFF$ 的 384KB 空间是被保留下来的，BIOS 就在其中。当 Intel 公司通过 80386 将可用物理地址空间扩大到 1MB 以上时，为保证对原有软硬件的向后兼容性，保留了原有对于最低 1MB 以内的地址的处理方式，它不支持分页和虚拟内存机制，即所谓的实模式。现代 PC 上因此将 $0x000A0000 \sim 0x00100000$ 这一段地址留成一个“空洞”，将内存划分成了两段，低 640K 称为“低段内存”，而将其余高地址部分称为扩展内存。新型的 x86 处理器型号至少有 32 位地址线，都可以支持超过 4G 的内存空间了，因此不得不使用类似的机制，在 4G 的内存部分再次留出一个空洞，用做 PCI 设备的预留空间。

2.4 BIOS

BIOS 是基本输入输出系统。在 PC 地址空间的 $0x000F0000 \sim 0x000FFFFF$ 的 64KB 区域。BIOS 负责处理系统基本的初始化任务, 如激活显示器、检查内存等。结束这些初始化任务之后, BIOS 从软盘、硬盘、光驱或者网络上载入操作系统, 并将控制权转移给操作系统。

在一个终端输入命令 `make qemu-gdb`, 进入 QEMU 的单步执行模式, 另一个终端启动 GDB。在 GDB 中可以发现 PC 启动的第一条指令:

```
[f000:fff0] 0xfffff0: ljmp    $0xf000,$0xe05b
```

表明 CS 寄存器被硬件初始化为 $0xf000$, IP 寄存器被初始化为 $0xffff0$, 即物理地址 $CS*16+IP=0xffff0$ 。这样一启动 BIOS 就可以立即取得控制权, 通过这条跳转语句就可以实现到地址 $0xf000*16+0xe05b=0xfe05b$ 的跳转, 真正执行 BIOS 的程序。此时还处在实模式中, 还没有进入保护模式。

3 Boot Loader

当 BIOS 找到启动的磁盘后, 便将 512 字节的启动扇区的内容装载到物理内存的 $0x7c00$ 到 $0x7dff$ 的位置, 紧接着再执行一个跳转指令将 CS 设置为 $0x0000$, IP 设置为 $0x7c00$, 这样便将控制权交给了 Boot Loader 程序。

在本实验中, Boot Loader 的源程序是由 `boot.S` (`boot/boot.S`) 的 AT&T 汇编程序与 `main.c` (`boot/main.c`) 的 C 程序组成。这两部分分别完成两个不同的功能。`boot.S` 主要是将处理器从实模式转换到 32 位的保护模式, 这是因为只有在保护模式中我们才能访问到物理内存高于 1MB 的空间。`main.c` 的主要作用是将内核的可执行代码从硬盘镜像中读入到内存中。另外, `obj/boot/boot.asm` 是当前 Boot Loader 在编译结束之后的反汇编代码, 可以通过此文件来跟踪整个 Boot Loader 的执行过程。

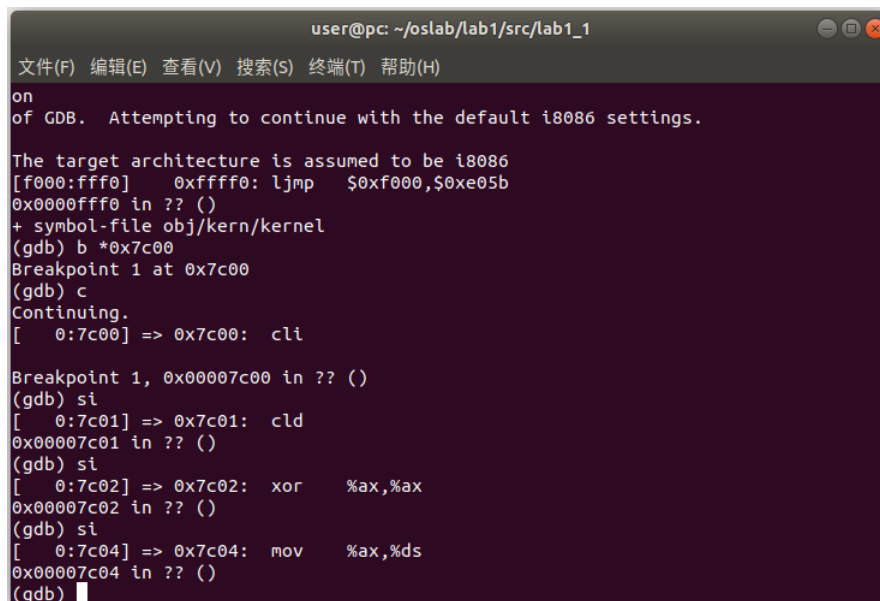
3.1 练习 2

3.1.1 题目原文

使用 GDB 的 debug 功能完成 Boot Loader 的追踪过程。b 命令设置断点, 如 `b *0x7c00` 表示在 $0x7c00$ 处设置断点; c 命令可执行到下一断点; si 命令是单步执行。

3.1.2 回答

根据题干，还是和前面一样我们可以在一个终端输入命令`make qemu-gdb`使 QEMU 进入单步调试模式，另一个终端打开 GDB。在 GDB 的窗口中时使用命令`b *0x7c00`在地址`0x7c00`处设置了断点，使用命令`c`执行到此断点处，这样就进入了 Boot Loader 程序。接下来使用命令`si`单步执行，由这些命令开始了对 Boot Loader 的追踪过程。



```
user@pc: ~/oslab/lab1/src/lab1_1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
on
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0] 0xfffff0: jmp $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) si
[ 0:7c01] => 0x7c01: cld
0x00007c01 in ?? ()
(gdb) si
[ 0:7c02] => 0x7c02: xor %ax,%ax
0x00007c02 in ?? ()
(gdb) si
[ 0:7c04] => 0x7c04: mov %ax,%ds
0x00007c04 in ?? ()
(gdb)
```

Figure 1: 追踪 Boot Loader

3.2 问题 1

3.2.1 题目原文

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.
```

确保你能回答以下几个问题：

- 1) At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- 2) What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?
- 3) Where is the first instruction of the kernel?
- 4) 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?

3.2.2 题目大意

在地址0x7c00设置断点，这是引导扇区将被加载的位置。继续执行直到那个断点。跟踪boot/boot.S中的代码，使用源代码和反汇编文件obj/boot/boot.asm来跟踪你在哪里。还可以使用 GDB 中的x/i命令来反汇编引导加载程序中的指令序列，并将原始引导加载程序源代码与obj/boot/boot.asm和 GDB 中的反汇编进行比较。

在boot/main.c中跟踪到bootmain(),然后进入readsect()。识别与readsect()中的每个语句相对应确切汇编指令。完成对readsect()其余部分跟踪,并回到bootmain(),并找到用来从盘读取内核其余部分的for循环的开始和结束的位置。找出循环完成后运行的代码，在那里设置一个断点，并继续执行到该断点。然后再走完 boot loader 程序的其余部分。

回答以下问题：

- 1) 处理器什么时候开始执行 32 位代码？如何完成的从 16 位到 32 位模式的切换？
- 2) 引导加载程序 boot loader 执行的最后一个指令是什么，加载的内核的第一个指令是什么？
- 3) 内核的第一个指令在哪里？
- 4) 引导加载程序 boot loader 如何决定为了从磁盘获取整个内核必须读取多少扇区？在哪里可以找到这些信息？

3.2.3 回答

(1)位于0x7c2d处的指令ljmp \$0x8, \$0x7c32控制跳转到 32 位代码,从位于0x7c32处的指令开始执行 32 位代码。

```

[ 0:7c23] => 0x7c23: mov    %cr0,%eax
0x00007c23 in ?? ()
(gdb)
[ 0:7c26] => 0x7c26: or     $0x1,%eax
0x00007c26 in ?? ()
(gdb)
[ 0:7c2a] => 0x7c2a: mov    %eax,%cr0
0x00007c2a in ?? ()
(gdb)
[ 0:7c2d] => 0x7c2d: ljmp   $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c32:      mov     $0x10,%ax
0x00007c32 in ?? ()
(gdb)

```

Figure 2: 追踪 Boot Loader

```

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
7c2d:      ea 32 7c 08 00 66 b8      ljmp    $0xb866,$0x87c32

```

Figure 3: obj/boot/boot.asm

在0x7c2a处，寄存器 CR0 的 0 位（即允许保护位 PE）被置为 1，从 16 位实模式改为 32 位保护模式。

```

[ 0:7c1e] => 0x7c1e: lgdtw  0x7c64
0x00007c1e in ?? ()
(gdb)
[ 0:7c23] => 0x7c23: mov    %cr0,%eax
0x00007c23 in ?? ()
(gdb)
[ 0:7c26] => 0x7c26: or     $0x1,%eax
0x00007c26 in ?? ()
(gdb)
[ 0:7c2a] => 0x7c2a: mov    %eax,%cr0
0x00007c2a in ?? ()
(gdb)
[ 0:7c2d] => 0x7c2d: ljmp   $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c32:      mov     $0x10,%ax

```

Figure 4: 追踪 Boot Loader

(2) 在指令0x7b6b处的call *0x10018是 boot loader 的最后一条指令，调用内核程序的入口。载入内核的第一条指令为下一条指令，即为存放在0x1000c处的movw \$0x1234, 0x472，在kern/entry.S中可以找到。

```

The target architecture is assumed to be i8086
[f000:fff0] 0xffff0: jmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x7d6b
Breakpoint 1 at 0x7d6b
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d6b: call *0x10018

Breakpoint 1, 0x00007d6b in ?? ()
(gdb) si
=> 0x10000c: movw $0x1234,0x472
0x0010000c in ?? ()
(gdb) █

```

Figure 5: 追踪 Boot Loader

从obj/boot/boot.asm中可以看到call *0x10018对应boot/main.c文件中, boot loader 最后的一行代码((void (*)(void)) (ELFHDR->e_entry))();, 这是从 ELF 文件头调用内核程序的入口。

```

7d5c:      83 c4 0c          add    $0xc,%esp
7d5f:      39 f3            cmp    %esi,%ebx
7d61:      72 e8            jb     7d4b <bootmain+0x40>
      ((void (*)(void)) (ELFHDR->e_entry))();
7d63:      ff 15 18 00 01 00 call   *0x10018|
}

```

Figure 6: obj/boot/boot.asm

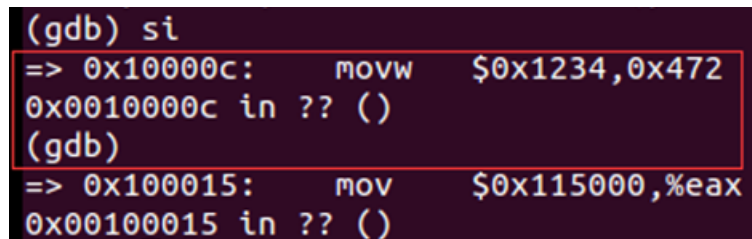
```

// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();

```

Figure 7: boot/main.c

(3)通过命令objdump -x obj/kern/kernel可以看到内核的起始地址为0x0010000c, 该处代码指令即为内核的第一条指令, 为movw \$0x1234, 0x472



```

(gdb) si
=> 0x10000c:    movw    $0x1234,0x472
0x0010000c in ?? ()
(gdb)
=> 0x100015:    mov     $0x115000,%eax
0x00100015 in ?? ()

```

Figure 8: 追踪 Boot Loader

(4) 操作系统中的段数量和每个段中包含的扇区信息在操作系统文件的 Program Header Table 中。

通过 ELF 文件头获取 Program Header Table，表中的每个表项对应于操作系统的一个段。每个 Program Header Table 记录三个信息用以描述段：p_pa (物理内存地址)，p_memsz (所占内存大小)，p_offset (相对文件的偏移地址)。根据这三个信息，对每个段从 p_offset (相对文件的偏移地址) 开始，读取 p_memsz (所占内存大小) 个字节的内容（需要根据扇区大小对齐），放入 p_pa (物理内存地址) 开始的内存中。

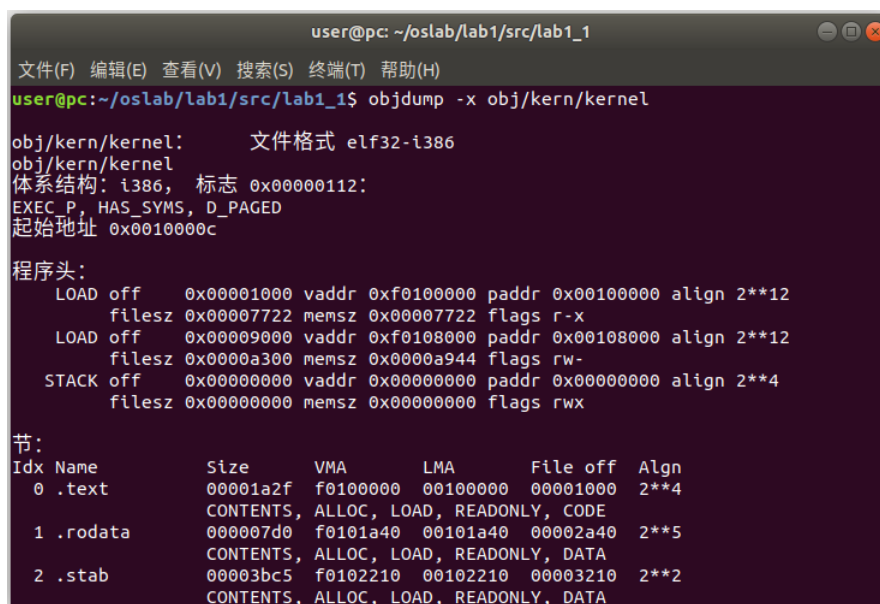
```

// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++)
    // p_pa is the load address of this segment (as well
    // as the physical address)
    readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

```

Figure 9: boot/main.c

利用 `objdump -x obj/kern/kernel` 命令可以查看内核程序的程序头，从这里可以得到需要读取的扇区数目。



```

user@pc: ~/oslab/lab1/src/lab1_1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
user@pc:~/oslab/lab1/src/lab1_1$ objdump -x obj/kern/kernel

obj/kern/kernel:      文件格式 elf32-i386
obj/kern/kernel
体系结构: i386, 标志 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x0010000c

程序头:
LOAD off  0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
      filesz 0x00007722 memsz 0x00007722 flags r-x
LOAD off  0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
      filesz 0x0000a300 memsz 0x0000a944 flags rw-
STACK off  0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
      filesz 0x00000000 memsz 0x00000000 flags rwx

节:
Idx Name          Size      VMA       LMA       File off  Algn
 0 .text          00001a2f  f0100000  00100000  00001000  2**4
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .rodata         000007d0  f0101a40  00101a40  00002a40  2**5
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .stab           00003bc5  f0102210  00102210  00003210  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA

```

Figure 10: objdump

3.3 载入内核

Boot Loader 除了将系统从实模式转换到保护模式之外还有一个重要的任务就是将内核的可执行程序 (ELF) 加载到内存。ELF 文件是 Linux 的可执行文件。包含载入信息 (loading information) 的头部和多个程序段 (program sections) 组成 (例如代码段 `.code` 和数据段 `.data` 等)。

通过输入命令: `objdump -h obj/kern/kernel` 可以获取 `kernel.ELF` 文件的段信息 (例如段的大小、地址等信息)。

通过输入命令: `objdump -f obj/kern/kernel` 可以获得 `kernel` 入口点的链接地址。



```

user@pc: ~/oslab/lab1/src/lab1_1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
user@pc:~/oslab/lab1/src/lab1_1$ objdump -h obj/kern/kernel

obj/kern/kernel:      文件格式 elf32-i386

节:
Idx Name              Size      VMA       LMA       File off  Algn
 0  .text              00001a1f  f0100000  00100000  00001000  2**4
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 1  .rodata            000007d0  f0101a20  00101a20  00002a20  2**5
    CONTENTS, ALLOC, LOAD, READONLY, DATA
 2  .stab              00003b89  f01021f0  001021f0  000031f0  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
 3  .stabstr           0000194d  f0105d79  00105d79  00006d79  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
 4  .data              0000a300  f0108000  00108000  00009000  2**12
    CONTENTS, ALLOC, LOAD, DATA
 5  .bss              00000644  f0112300  00112300  00013300  2**5
    ALLOC
 6  .comment           0000002b  00000000  00000000  00013300  2**0
    CONTENTS, READONLY
user@pc:~/oslab/lab1/src/lab1_1$ objdump -f obj/kern/kernel

obj/kern/kernel:      文件格式 elf32-i386
体系结构: i386, 标志 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x0010000c
user@pc:~/oslab/lab1/src/lab1_1$

```

Figure 11: 查看 kernel 的段信息和入口点的链接地址

由此也可以看出kernel的入口点的链接地址是0x0010000c。注意到，上图中的VMA是链接地址（link address），是程序自己假设在内存中的存放的位置，即编译器编译时认定程序将会存放在以链接地址开始的连续内存空间；而LMA是载入地址（load address）实际指的是程序真正存放的地址。在实模式下两者的值一般是一样的。但在这里看起来不一样的原因是在保护模式下 JOS 中内核实际被载入的地址被映射到虚拟地址的高地址处（0xf0000000~0xffffffff），这样做的一个目的是给用户程序留下足够大的虚拟地址空间。例如从obj/kern/kernel.asm中可以发现，内核的第一条指令对应的地址0xf010000c其实是VMA，而从这里看到的地址0x0010000c是LMA，这在2.2节中的kerninfo中也能看出来。

3.3.1 对 pointer.c 的理解

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void f(void)
5 {
6     int a[4]; //设数组a在某一次运行的首地址为: 0x0061FDF0
7     int *b = malloc(16); //动态分配4*4=16字节内存

```

```
8  int *c; //此时c未初始化, 应在下面的printf语句打印随机值
9  int i;
10
11  printf("1: a = %p, b = %p, c = %p\n", a, b, c);
12
13  c = a; //将数组a的首地址赋给指针变量c
14  for (i = 0; i < 4; i++)
15      a[i] = 100 + i;
16  c[0] = 200; //用指针变量c来间接访问数组a
17  printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
18      a[0], a[1], a[2], a[3]); //200 101 102 103
19
20  c[1] = 300;
21  *(c + 2) = 301; //c[2]
22  3[c] = 302; //在C语言中, 3[c]与c[3]等价
23  printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
24      a[0], a[1], a[2], a[3]); //200 300 301 302
25
26  c = c + 1; //c指向a[1]
27  *c = 400; //a[1]=400
28  printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
29      a[0], a[1], a[2], a[3]); //200 400 301 302
30
31  c = (int *) ((char *) c + 1); //c指向的地址为0x0061FDF5
32  *c = 500; //500=0x000001F4
33  printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
34      a[0], a[1], a[2], a[3]);
35  //常见的x86体系结构使用小端模式
36  //0x000000C8 (200) 0x0001f490 (128144) 0x00000100 (256) 302
37
38  b = (int *) a + 1; //b指向a[1], 即0x0061FDF4
39  c = (int *) ((char *) a + 1); //c指向的地址为0x0061FDF1
40  printf("6: a = %p, b = %p, c = %p\n", a, b, c);
41 }
42
43 int main(int ac, char **av)
44 {
45     f();
46     return 0;
47 }
```

对于程序中第 5 个输出, 改变变量c指向的地址前以及c改变后并对c指向内容进行

赋值操作后的图解（使用小端模式存储）：

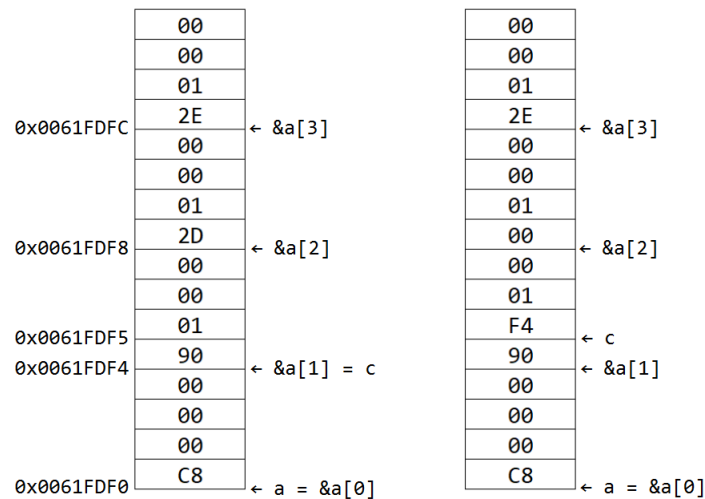


Figure 12: 图解

4 The Kernal

4.1 问题 2

4.1.1 题目原文

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      }
```

4.1.2 题目大意

1. 解释printf.c和console.c之间的接口。具体来说, console.c导出了什么函数? printf.c如何使用这些函数?
2. 从console.c解释以下内容:

```
1      if (crt_pos >= CRT_SIZE) {
2          int i;
3              memmove(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      }
```

4.1.3 回答

第1问中printf.c和console.c的关系:

kern/printf.c用来实现在内核控制台打印字符串,这依托于kern/console.c提供的与硬件进行交互的接口以及lib/printfmt.c对格式化输出字符操作的定义。

其中,kern/printf.c的cprintf()实现在控制台输出字符串,调用了vcprintf(),vcprintf()又调用了lib/printfmt.c的vprintfmt()函数实现格式化输出,vprintfmt()又调用了kern/printf.c中的putch()函数,putch()又调用了kern/console.c的cputchar()函数。cputchar()函数又调用了同样在kern/console.c中的cons_putc()函数,cons_putc()函数又调用了同样在kern/console.c的三个函数serial_putc()、lpt_putc()、cga_putc()。serial_putc()和lpt_putc()几乎就是与I/O端口直接交互了,都是提供输出字符前的准备工作,serial_putc()能够保证串行输出字符,lpt_putc()可以保证并行端口的输出;cga_putc()则是真正地设置打印在屏幕上的字符的显示属性。下图直观地表示了这种关系:

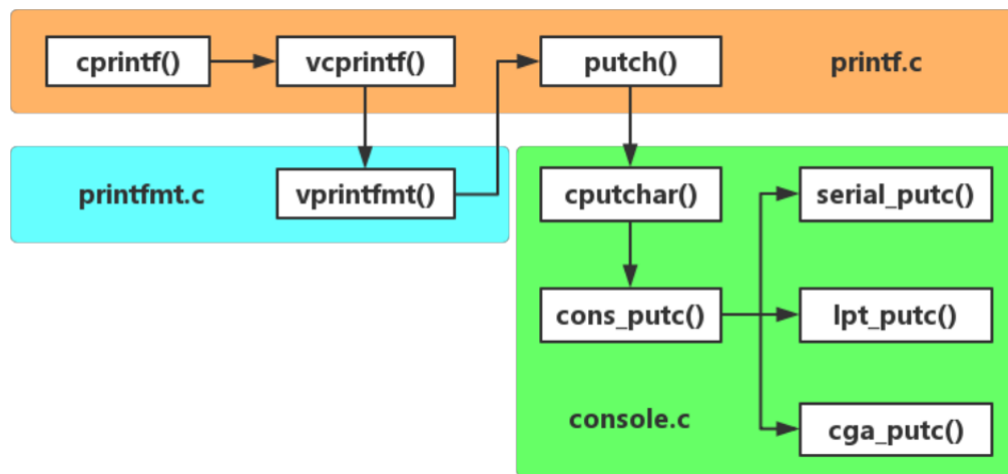


Figure 13: 关系图

对第 2 问程序片段的解释：

此段代码在 `cga_putc()` 中，作用是如果要在屏幕上显示的内容满了，达到或超出了屏幕能显示的范围，后续的内容显示不下了，就把屏幕的内容向上滚动一行，以使后续的内容可以显示在屏幕的最后一行。分析如下：

```

1 static void
2 cga_putc(int c)
3 {
4     // if no attribute given, then use black on white
5     if (!(c & ~0xFF))
6         c |= 0x0700;
7
8     .....
9
10    // What is the purpose of this?
11    if (crt_pos >= CRT_SIZE) {
12        int i;
13
14        //将屏幕显示部分的第二行到最后一行整体往上挪一行
15        memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(
uint16_t));
16        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++) //最后一行内容清空
17            crt_buf[i] = 0x0700 | ' ';
18        crt_pos -= CRT_COLS; //将屏幕光标向上挪动一行
19    }
20    .....

```

先解释出现的四个标识符：

`crt_buf`数组就是屏幕要显示的内容的缓冲区，`CRT_SIZE`是屏幕至多能显示的字符数，`crt_pos`代表屏幕当前的光标位置（可以看作是`crt_buf`中最后一个字符的位置），`CRT_COLS`代表光标实际在屏幕上移动一行需要挪动的距离（显示器每一行的字符数）。

其中，

```
memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS)
* sizeof(uint16_t));
```

是把以`crt_buf + CRT_COLS`为首地址的内容拷贝(`CRT_SIZE - CRT_COLS`)个字符到首地址`crt_buf`处，即把屏幕从第二行开始到最后一行的内容拷贝到屏幕最开始的地方，对应的显示效果就是第一行的内容没有了，后面一行覆盖了前面一行的内容。这里的第一个参数是`crt_buf`数组的首地址，代表屏幕要显示的第一个字符，第二个参数`crt_buf + CRT_COLS`即指向屏幕第二行的开始，第三个参数(`CRT_SIZE - CRT_COLS`)*`sizeof(uint16_t)`即为屏幕内容从第二行到最后一行字符所占的字节数。`for`循环的作用是把最后一行的内容清空，以留出空来显示后续的字符。从上述的第5行第6行可以看出，`crt_buf[i] = 0x0700 | ''`；就是把缓冲区这个区域用空格（黑色底）填充。`crt_pos -= CRT_COLS`；就是把光标向上挪动一行，也就是挪动到最后一行开始的位置，以使屏幕可以显示后续的字符。

事实上，每个字符输出到屏幕上时都会调用这个函数，先设置该字符输出到屏幕上的属性，再作`crt_pos >= CRT_SIZE`的判断，即实际要显示的字符数是否达到了屏幕至多能显示的字符数。

4.2 作业 1

4.2.1 题目原文

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.

4.2.2 题目大意

补齐打印八进制数字代码的实现，格式控制符为`%o`。

4.2.3 回答

问题 2 中已解释kern/printf.c、kern/console.c和lib/printfmt.c三个文件的作用。在文件lib/printfmt.c中仿照十进制更改代码，使能支持%o输出八进制：

```
// unsigned decimal
case 'u':
    num = getuint(&ap, lflag);
    base = 10;
    goto number;
```

Figure 14: lib/printfmt.c 中实现十进制输出的代码

<pre>// (unsigned) octal case 'o': // Replace this with your code. putch('X', putdat); putch('X', putdat); putch('X', putdat); break;</pre> <p>(a) 修改前</p>	<pre>// (unsigned) octal case 'o': // Replace this with your code. //从ap指向的可变字符串中获取输出的值 num = getuint(&ap, lflag); //设置基数为8 base = 8; goto number;</pre> <p>(b) 修改后</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 15: lib/printfmt.c 中实现八进制输出的代码

注意到原先内核输出的第一行为：

6828 decimal is XXX octal!

对应kern/init.c中的i386_init()函数，此句用来测试十进制数 6828 对应八进制数的输出：

```

void
i386_init(void)
{
    extern char edata[], end[];

    // Before doing anything else, complete the ELF loading process.
    // Clear the uninitialized global data (BSS) section of our program.
    // This ensures that all static/global variables start out zero.
    memset(edata, 0, end - edata);

    // Initialize the console.
    // Can't call cprintf until after we do this!
    cons_init();

    cprintf("6828 decimal is %o octal!\n", 6828);

    // Test the stack backtrace function (lab 1 only)
    test_backtrace(5);

    // Drop into the kernel monitor.
    while (1)
        monitor(NULL);
}

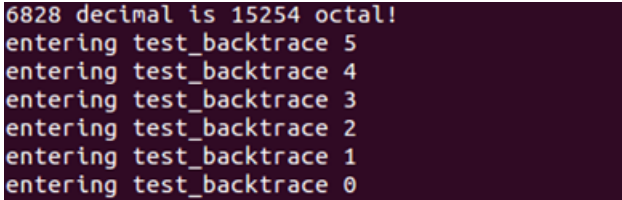
```

Figure 16: kern/init.c

经过这里的修改，内核第一行的输出已变为：

6828 decimal is 15254 octal!

输出八进制数 15254，表明成功实现了八进制的输出。



```

6828 decimal is 15254 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

```

Figure 17: 内核输出

4.3 堆栈

在进程的地址空间中，栈是由操作系统来维护的一块内存空间，存储一些局部变量、函数调用的参数等。它是由高地址向低地址增长的。涉及到的三个重要的寄存器：`%eip`指向要执行的下一条指令的地址，`%ebp`指向栈底（在高地址处），`%esp`指向栈顶（在低地址处）。

4.4 练习 3

4.4.1 题目原文

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?

4.4.2 题目大意

在`obj/kern/kernel.asm`中找到`test_backtrace`函数的地址,在其中设置一个断点,并检查在内核启动后每次这个函数被调用时会发生什么。每一级的`test_backtrace`在递归调用时,会在栈上压入多少个 32 位的字,这些字的内容是什么?

4.4.3 回答

从`obj/kern/kernel.asm`中找到`test_backtrace()`函数的入口地址为`0xf0100040`,打开 GDB,输入命令**b *0xf0200040**在此地址处设置一个断点,再输入命令**c**使内核程序执行到这个断点处。下面是`obj/kern/kernel.asm`中`test_backtrace()`被反汇编的部分。

```

1 // Test the stack backtrace function (lab 1 only)
2 void
3 test_backtrace(int x)
4 {
5 f0100040: 55                push    %ebp
6 f0100041: 89 e5            mov     %esp,%ebp
7 f0100043: 53              push    %ebx
8 f0100044: 83 ec 14        sub     $0x14,%esp
9 f0100047: 8b 5d 08        mov     0x8(%ebp),%ebx
10    cprintf("entering test_backtrace %d\n", x);
11 f010004a: 89 5c 24 04      mov     %ebx,0x4(%esp)
12 f010004e: c7 04 24 20 1a 10 f0 movl    $0xf0101a20,(%esp)
13 f0100055: e8 45 09 00 00   call    f010099f <cstdio>
14    if (x > 0)
15 f010005a: 85 db          test    %ebx,%ebx
16 f010005c: 7e 0d          jle     f010006b <test_backtrace+0x2b>

```

```

17     test_backtrace(x-1);
18 f010005e: 8d 43 ff          lea    -0x1(%ebx),%eax
19 f0100061: 89 04 24          mov    %eax,(%esp)
20 f0100064: e8 d7 ff ff ff    call   f0100040 <test_backtrace>
21 f0100069: eb 1c            jmp    f0100087 <test_backtrace+0x47>
22     else
23     mon_backtrace(0, 0, 0);
24 f010006b: c7 44 24 08 00 00 00 movl    $0x0,0x8(%esp)
25 f0100072: 00
26 f0100073: c7 44 24 04 00 00 00 movl    $0x0,0x4(%esp)
27 f010007a: 00
28 f010007b: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
29 f0100082: e8 01 07 00 00    call   f0100788 <mon_backtrace>
30     cprintf("leaving test_backtrace %d\n", x);
31 f0100087: 89 5c 24 04       mov    %ebx,0x4(%esp)
32 f010008b: c7 04 24 3c 1a 10 f0 movl    $0xf0101a3c,(%esp)
33 f0100092: e8 08 09 00 00    call   f010099f <cstdio>
34 }
35 f0100097: 83 c4 14          add    $0x14,%esp
36 f010009a: 5b               pop    %ebx
37 f010009b: 5d               pop    %ebp
38 f010009c: c3              ret

```

由此可见，`test_backtrace()`的动作就是每次调用这个函数先打印一行进入到此函数的信息和当前调用此函数的参数值，然后该参数值减 1，再以现在的参数值递归调用此函数，直到参数变为 0 停止递归调用，参数变为 0 时会调用`mon_backtrace()`函数来打印栈回溯的信息。最后离开这个函数前再打印离开此函数的信息和当前调用此函数的参数值的参数值。递归调用`test_backtrace()`使得最先进入的程序，即参数最大的函数调用最后退出。

这些内容会被压入栈中：

- 函数入口处先让`%ebp`入栈，保存调用者的`%ebp`，共 1 个 32 位的字。
- 第 7 行`%ebx`入栈，由后面的代码可知，`%ebx`的值会被修改（即调用者调用`test_backtrace()`所传入的参数，`0x8(%ebp)`即为该参数在内存中的位置），作为调用`cprintf()`的参数之一，共 1 个 32 位的字。
- 第 8 行预留了`0x14`个即 20 个 byte 的空间作为临时变量储存（共 5 个 32 位的字），包括当前函数作为调用者调用其他函数时，给被调用的函数的参数，也放在这个空间中。

- 另外，在执行`call`指令是时，`%eip`的值也会入栈（即函数返回地址的值，函数返回之后要执行的下一条指令），共 1 个 32 位的字。

综上，每次调用，一共会有 $1+1+5+1=8$ 个 32 位的字被压入栈中。

4.5 作业 2

4.5.1 题目原文

```
查看test_backtrace()的C代码（kern/init.c中），完成其中的mon_backtrace()，
mon_backtrace的原型已经在kern/monitor.c中。
You can do mon_backtrace() entirely in C. 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:
Warning: read_ebp()（较为底层的函数，返回值为当前ebp寄存器的值）
Display format:
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2
    00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28
    00000061
  ...
```

4.5.2 题目大意

补齐文件`kern/monitor.c`中的实现堆栈回溯功能的函数`mon_backtrace()`的实现，函数按照上述格式输出。

4.5.3 回答

在进程的地址空间中，由栈来维护的，在栈底寄存器`%ebp`与栈顶寄存器`%esp`之间的空间是当前函数的栈帧。调用者与被调用者的栈帧结构显示如下图：

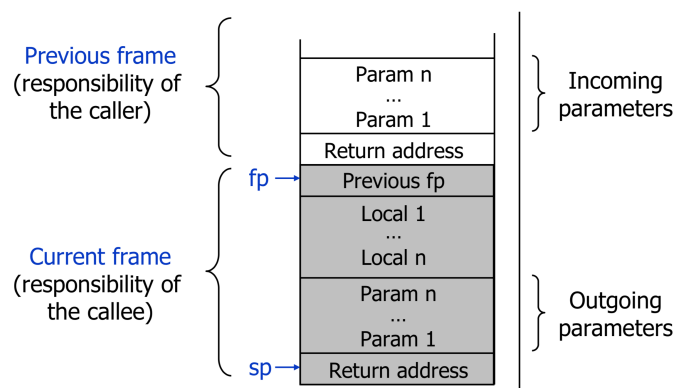


Figure 18: 栈帧结构

按照上图，代码实现的思路如下：可以先调用题干所提到的`read_ebp()`函数获取当前函数`mon_backtrace()`的栈底指针`%ebp`，把它存入变量`ebp`。`%ebp`就像一个链，可以通过这个获取到的`%ebp`值来向前回溯寻找各个调用者的栈帧，然后就可以通过`%ebp`来获取各个调用者的`%ebp`，`%eip`和调用下一个函数所传的参数。如下图：

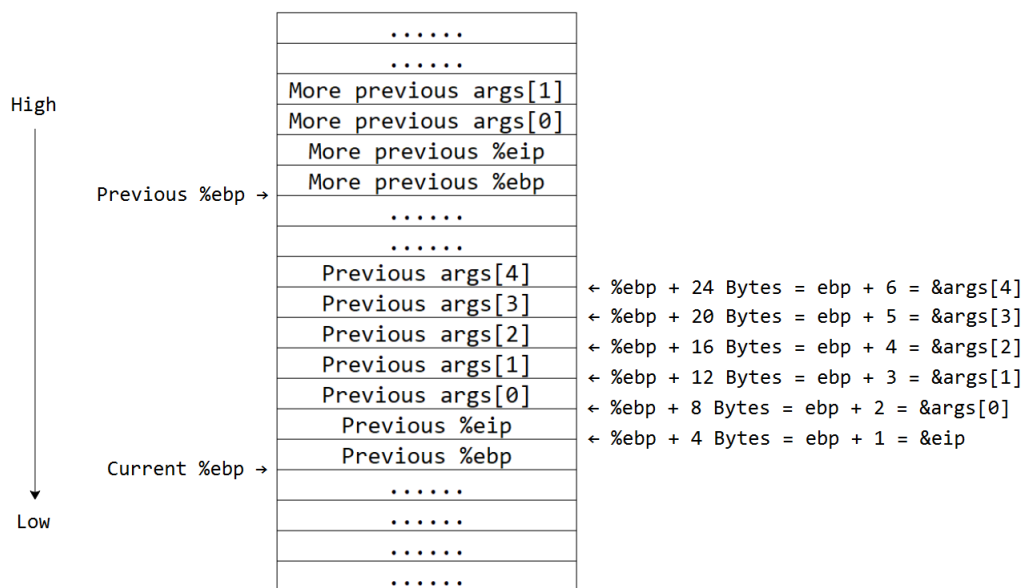


Figure 19: 思路图

因此，`mon_backtrace()`的代码实现如下：

```

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    // Read ebp of mon_backtrace()
    unsigned int *ebp = (unsigned int *) read_ebp();
    // The first five args of the current function
    unsigned int args[5];
    cprintf("Stack backtrace:\n");
    while(ebp) {
        unsigned int eip = (unsigned int) *(ebp + 1);
        args[0] = (unsigned int) *(ebp + 2);
        args[1] = (unsigned int) *(ebp + 3);
        args[2] = (unsigned int) *(ebp + 4);
        args[3] = (unsigned int) *(ebp + 5);
        args[4] = (unsigned int) *(ebp + 6);
        cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n", ebp, eip,
                args[0], args[1], args[2], args[3], args[4]);
        ebp = (unsigned int *) *ebp;
    }
    return 0;
}

```

Figure 20: mon_backtrace()

其中, `cprintf()` 是在内核控制台的屏幕上输出字符串的函数, 在 `kern/printf.c` 中定义。格式控制符 `%08x` 表示控制输出的字符数为 8 个字符, 以右对齐的方式输出十六进制数, 左边的空白以字符 `'0'` 补齐。这里 `while()` 循环的循环变量为 `ebp`, 取 `ebp` 所指向地址的内容即为上一个函数的 `%ebp`。数据类型使用 `unsigned int` 和 `unsigned int *` 是因为内存的地址值都是无符号 16 进制整数, 事实上 `%ebp`、`%eip` 的值都是地址值, 这样通过 `unsigned int*` 取到的内容也需要强制转换成 `unsigned int`。当然, 函数的参数值未必就是 `unsigned int` 类型的, 这里为方便起见都按 `unsigned int` 类型处理。

循环终止的条件为 `ebp==0` 的原因在 `kern/entry.S` 中: 在执行 `i386_init()` 之前会执行指令 `movl $0x0,%ebp`, 作用就是将 `%ebp` 的值初始化为 0, 这样一旦开始调试 C 代码, 堆栈回溯将正确终止。因此 0 即为 `%ebp` 的初始值, 即回溯的终点。

```

relocated:

    # Clear the frame pointer register (EBP)
    # so that once we get into debugging C code,
    # stack backtraces will be terminated properly.
    movl    $0x0,%ebp                # nuke frame pointer

    # Set the stack pointer
    movl    $(bootstacktop),%esp

    # now to C code
    call    i386_init

```

Figure 21: kern/entry.S

完成此题后内核的输出:

```
6828 decimal is 15254 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
Stack backtrace:
  ebp f010ff18  eip f0100087  args 00000000 00000000 00000000 00000000
    f0100959
  ebp f010ff38  eip f0100069  args 00000000 00000001 f010ff78 00000000
    f0100959
  ebp f010ff58  eip f0100069  args 00000001 00000002 f010ff98 00000000
    f0100959
  ebp f010ff78  eip f0100069  args 00000002 00000003 f010ffb8 00000000
    f0100959
  ebp f010ff98  eip f0100069  args 00000003 00000004 00000000 00000000
    00000000
  ebp f010ffb8  eip f0100069  args 00000004 00000005 00000000 00010094
    00010094
  ebp f010ffd8  eip f01000ea  args 00000005 00001aac 00000644 00000000
    00000000
  ebp f010fff8  eip f010003e  args 00111021 00000000 00000000 00000000
    00000000
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>
```

4.6 挑战作业

4.6.1 题目原文

(原 Lab 1 的 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?

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 0000
ffff
```

```
    kern/entry.S:70: <unknown>+0
```

K>

4.6.2 题目大意

1. 补全函数`debuginfo_eip`对`stab_binsearch`的调用来查找地址的行号的实现。
2. 向内核监视器添加`backtrace`命令。
3. 扩展`mon_backtrace`的实现，按照上述格式完成对`mon_backtrace`的函数名，文件名等信息的打印。

4.6.3 回答

首先，必须要清楚**Stab**是什么。在 GCC 编译器中，汇编器将**Stab**中的信息默认情况下放置在符号表和字符串表中的符号信息中。链接器将**.o**文件合并为一个可执行文件**.ELF**，其中包含一个符号表和一个字符串表。因此，**Stab**在 Linux 的可执行文件（**.ELF**）中就是符号表，可为调试器提供程序调试信息的来源。**Stabstr**就是符号表所对应的字符串表，打印的效果是符号表项的字符串值。

在kern/kernel.ld,找到了.stab段和.stabstr段的定义。其中__STAB_BEGIN__、__STAB_END__、__STABSTR_BEGIN__、__STABSTR_END__在kern/kern.ld中定义,分别是.stab和.stabstr段开始与结束的地址:

```
/* Include debugging information in kernel memory */
.stab : {
    PROVIDE(__STAB_BEGIN__ = .);
    *(.stab);
    PROVIDE(__STAB_END__ = .);
    BYTE(0)          /* Force the linker to allocate space
                       for this section */
}

.stabstr : {
    PROVIDE(__STABSTR_BEGIN__ = .);
    *(.stabstr);
    PROVIDE(__STABSTR_END__ = .);
    BYTE(0)          /* Force the linker to allocate space
                       for this section */
}
```

Figure 22: kern/kernel.ld 中的段定义

在inc/stab.h中又找到了Stab的结构体定义:

```
// Entries in the STABS table are formatted as follows.
struct Stab {
    uint32_t n_strx;          // index into string table of name
    uint8_t n_type;           // type of symbol
    uint8_t n_other;          // misc info (usually empty)
    uint16_t n_desc;          // description field
    uintptr_t n_value;        // value of symbol
};
```

Figure 23: inc/stab.h 中的 Stab 结构体定义

Stab结构体的字段意义如下:

- **n_strx**: 该符号表项的字符串在字符串表 stabstr 中的偏移 (相对于字符串表首地址)。
- **n_type**: 该符号表项描述的符号类型。
- **n_other**: 此项一般为空。
- **n_desc**: 表示在源文件中的行号。
- **n_value**: 相应的内容运行时的地址 (如果对于SLINE表项来说还可能是相对于函数入口地址的偏移)。

输入命令`objdump -h obj/kern/kernel`可以获取可执行文件`kernel`的段信息，其中有段的大小，段起始地址的虚拟地址（VMA）、装载地址（LMA）等信息。此图在3.3节中，这里不再重复列出。

输入命令`objdump -G obj/kern/kernel`查看可执行文件`kernel`的符号表信息:（内容很多，就不完全展示了）

```
obj/kern/kernel:      文件格式 elf32-i386

.stab 节的内容:

Symnum n_type n_othr n_desc n_value  n_strx String
-1      HdrSym 0      1269   0000194c 1
0       SO     0      0      f0100000 1      {standard input}
1       SOL    0      0      f010000c 18     kern/entry.S
2       SLINE  0      44     f010000c 0
3       SLINE  0      57     f0100015 0
4       SLINE  0      58     f010001a 0
5       SLINE  0      60     f010001d 0
6       SLINE  0      61     f0100020 0
7       SLINE  0      62     f0100025 0
8       SLINE  0      67     f0100028 0
9       SLINE  0      68     f010002d 0
10      SLINE  0      74     f010002f 0
11      SLINE  0      77     f0100034 0
12      SLINE  0      80     f0100039 0
13      SLINE  0      83     f010003e 0
14      SO     0      2      f0100040 31     kern/entrypgdir.c
15      OPT    0      0      00000000 49     gcc2_compiled.
16      LSYM   0      0      00000000 64     int:t(0,1)=r(0,1)
      ; -2147483648; 2147483647;
17      LSYM   0      0      00000000 106    char:t(0,2)=r(0,2);0;127;
18      LSYM   0      0      00000000 132    long int:t(0,3)=r(0,3)
      ; -2147483648; 2147483647;
19      LSYM   0      0      00000000 179    unsigned
.....
102     SLINE  0      20     00000057 0
103     RSYM   0      0      00000003 2747   x:r(0,1)
104     FUN     0      0      f010009d 2756   i386_init:F(0,18)
105     SLINE  0      24     00000000 0
106     SLINE  0      30     00000006 0
```

```

107    SLINE  0      34      00000028 0
108    SLINE  0      36      0000002d 0
109    SLINE  0      39      00000041 0
110    SLINE  0      43      0000004d 0
111    FUN    0      0       f01000f8 2774  _panic:F(0,18)
.....

```

其中,表头部分`n_type`、`n_othr`、`n_desc`、`n_value`、`n_strx`与上述的`Stab`结构体字段一一对应。`Symnum`为符号表项在符号表中的行号。类型名称`n_type`中,`S0`在`String`字段中为函数对应的文件名,`SLINE`为源文件中引用此符号的行号,`FUN`是函数名称。

输入命令:

```
gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -
DJOS_KERNEL -gstabs -c -S kern/init.c
```

可以看到`init.c`被编译出来的`init.s`文件:



```

user@pc: ~/oslab/lab1/src/lab1_1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
test_backtrace:
    .stabsn 68,0,13,.LM0-.LFBB1
.LM0:
.LFBB1:
.LFB0:
    .cfi_startproc
    pushq %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    .stabsn 68,0,14,.LM1-.LFBB1
.LM1:
    movl %edi, %esi
    .stabsn 68,0,13,.LM2-.LFBB1
.LM2:
    movl %edi, %ebx
    .stabsn 68,0,14,.LM3-.LFBB1
.LM3:
    leaq .LC0(%rip), %rdi
    xorl %eax, %eax
    call cprintf@PLT
    .stabsn 68,0,15,.LM4-.LFBB1
.LM4:
    testl %ebx, %ebx
    jg .L6
    .stabsn 68,0,18,.LM5-.LFBB1
.LM5:
    xorl %edx, %edx
    xorl %esi, %esi
    xorl %edi, %edi
    call mon_backtrace@PLT
.L3:
    .stabsn 68,0,19,.LM6-.LFBB1

```

Figure 24: `init.s`

其中找到了函数`test_backtrace()`的汇编代码。对`Stab`进行操作的代码在`kern/kdebug.c`中,其中有函数`stab_binsearch()`和`debuginfo_eip()`。由注释可知,函数`stab_binsearch(*stabs, *region_left, *region_right, type, addr)`用来

查找包含`addr`所指定的地址的符号表项, `*region_left`和`*region_right`在符号表的行号上限制了查找范围为`[*region_left, *region_right]` (查找的原理是二分查找), `type`为要查找的符号表项的类型 (例如`N_FUN`是函数表项`FUN`, `N_SO`是文件表项`SO`), 参数`stabs`应传入符号表的起始地址`__STABS_BEGIN__`。函数返回以后会修改参数`*region_left`和`*region_right`所对应的实参的内容, `*region_left`被修改为`n_value <= addr`且离`addr`最近的同类型符号表项行号, `*region_right`被修改为`n_value > addr`且离`addr`最近的同类型符号表项行号减1。如果返回的`*region_left > *region_right`则没有找到参数`addr`所对应的表项。

`debuginfo_eip(addr, info)`则是查找该地址`addr` (可以是`eip`) 所对应的调试信息`info`, 并完善`info`中的信息, 最终的信息在参数`info`中被返回。如果找到了相应的信息, 则函数的返回值为0, 找不到则为-1。`info`参数对应的结构体`Eipdebuginfo`在`kern/kdebug.h`中被定义:

```
// Debug information about a particular instruction pointer
struct Eipdebuginfo {
    const char *eip_file;           // Source code filename for EIP
    int eip_line;                   // Source code linenumber for EIP

    const char *eip_fn_name;        // Name of function containing EIP
                                    // - Note: not null terminated!
    int eip_fn_namelen;             // Length of function name
    uintptr_t eip_fn_addr;          // Address of start of function
    int eip_fn_narg;                // Number of function arguments
};
```

Figure 25: 结构体 `Eipdebuginfo`

各个字段的意义在注释中。在`mon_backtrace()`中要打印的文件名、函数名、行号信息将从这个结构体中提取。

在上图所对应符号表中, 可以发现行号按照代码段指令地址 (`n_value`) 升序排列。对于`SO`表项来说是按照这些文件在可执行文件中被链接的地址排列的, 而`FUN`表项是按照这些函数在文件中被定义的地址 (即函数入口地址) 排列的。当然, 同一个文件中函数所对应的`FUN`表项一定在该文件所对应的`SO`表项之后, 下一个`SO`表项之前。此思路被用于在函数`debuginfo_eip()`使用`stab_binsearch()`查找`FUN`表项和`SO`表项。

在函数`debuginfo_eip()`中, 对`info`的查找则是先查找`addr`对应的`SO`表项, 找到`SO`表项之后查找范围被缩小, 再根据这个被缩小的查找范围查找 `FUN` 表项, 最后在更小的查找范围查找`SLINE`表项。原先对`SLINE`表项的查找代码被省略, 这是需要补齐的地方。

因此, 根据注释以及函数`debuginfo_eip()`中对`SO`表项和`FUN`表项查找的代码, 可以仿照写出对`SLINE`表项查找的代码 (观察代码`addr -= info->eip_fn_addr;`可知现在的`addr`参数可能已经变成了相对于函数入口地址的偏移, 有的`SLINE`表项的`n_value`就是这个偏移):

```

// Search within [lline, rline] for the line number stab.
// If found, set info->eip_line to the right line number.
// If not found, return -1.
//
// Hint:
//     There's a particular stabs type used for line numbers.
//     Look at the STABS documentation and <inc/stab.h> to find
//     which one.
// Your code here.
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);

if (lline <= rline) {
    info->eip_line = stabs[lline].n_desc;
}
else {
    return -1;
}

```

Figure 26: 补齐 debuginfo_eip() 中查找 SLINE 的代码

其中实参 `stabs` 在函数 `debuginfo_eip()` 中被初始化为符号表的起始地址:

```

// Find the relevant set of stabs
if (addr >= ULIM) {
    stabs = __STAB_BEGIN__;
    stab_end = __STAB_END__;
    stabstr = __STABSTR_BEGIN__;
    stabstr_end = __STABSTR_END__;
} else {
    // Can't search for user-level addresses yet!
    panic("User address");
}

```

Figure 27: 补齐 debuginfo_eip() 中查找 SLINE 的代码

`info` 的内容也被初始化:

```

// Initialize *info
info->eip_file = "<unknown>";
info->eip_line = 0;
info->eip_fn_name = "<unknown>";
info->eip_fn_namelen = 9;
info->eip_fn_addr = addr;
info->eip_fn_narg = 0;

```

Figure 28: 补齐 debuginfo_eip() 中查找 SLINE 的代码

接下来, 就是补齐 `mon_backtrace()` 的代码了:

```

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    // Read ebp of mon_backtrace()
    unsigned int *ebp = (unsigned int *) read_ebp();
    // The first five args of the current function
    unsigned int args[5];
    cprintf("Stack backtrace:\n");
    while(ebp) {
        unsigned int eip = (unsigned int) *(ebp + 1);
        args[0] = (unsigned int) *(ebp + 2);
        args[1] = (unsigned int) *(ebp + 3);
        args[2] = (unsigned int) *(ebp + 4);
        args[3] = (unsigned int) *(ebp + 5);
        args[4] = (unsigned int) *(ebp + 6);
        cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\n", ebp, eip,
                args[0], args[1], args[2], args[3], args[4]);
        struct Eipdebuginfo info;
        debuginfo_eip((uintptr_t) eip, &info);
        cprintf("          %s:%d: %.*s+%d\n", info.eip_file, info.eip_line,
                info.eip_fn_namelen, info.eip_fn_name, eip - info.eip_fn_addr);
        ebp = (unsigned int *) *ebp;
    }
    return 0;
}

```

Figure 29: 修改过的 mon_backtrace()

其中，由于info.eip_fn_name中除了函数名以外还附带了其他信息，需要通过函数名长度info.eip_fn_namelen得到纯函数名，方式是使用输出控制符%.s来打印从头开始的指定长度的字符串，有两个参数：info.eip_fn_namelen指定字符串的长度（对应输出控制符的*），info.eip_fn_name为源字符串（对应输出控制符的s）。eip相对于函数入口地址的偏移地址由eip的值减去函数入口地址info.eip_fn_addr得到。

最后，需要在kern/monitor.c中为内核添加调用mon_backtrace()函数的backtrace命令，如下图所示：

```

static struct Command commands[] = {
    { "help", "Display this list of commands", mon_help },
    { "kerninfo", "Display information about the kernel", mon_kerninfo },
    { "backtrace", "Display information about the stack backtrace", mon_backtrace },
};

```

Figure 30: 在命令监视器中添加新命令 backtrace

最后的输出效果如下：

```

6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2

```

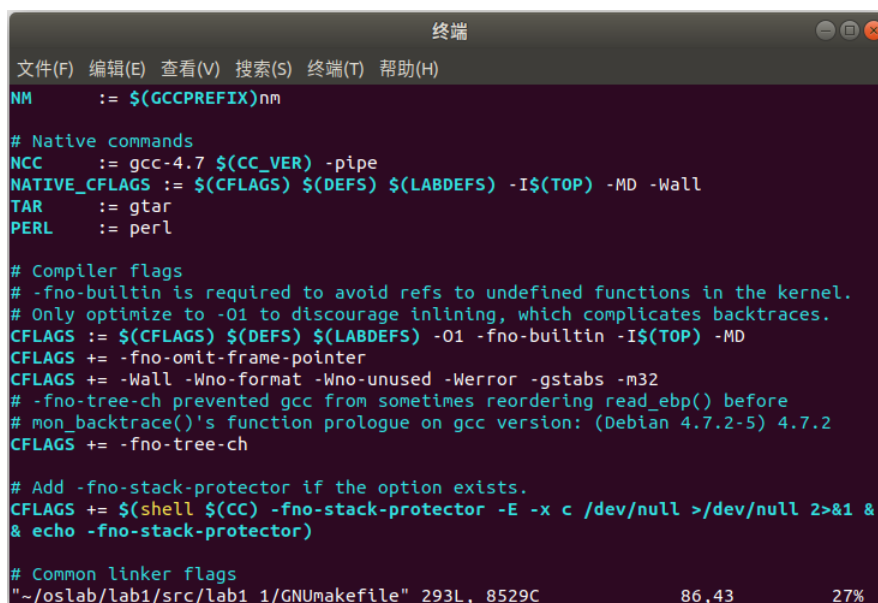
```
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
  ebp f010ff18  eip f0100087  args 00000000 00000000 00000000 00000000
    f010095b
      kern/init.c:19: test_backtrace+71
  ebp f010ff38  eip f0100069  args 00000000 00000001 f010ff78 00000000
    f010095b
      kern/init.c:16: test_backtrace+41
  ebp f010ff58  eip f0100069  args 00000001 00000002 f010ff98 00000000
    f010095b
      kern/init.c:16: test_backtrace+41
  ebp f010ff78  eip f0100069  args 00000002 00000003 f010ffb8 00000000
    f010095b
      kern/init.c:16: test_backtrace+41
  ebp f010ff98  eip f0100069  args 00000003 00000004 00000000 00000000
    00000000
      kern/init.c:16: test_backtrace+41
  ebp f010ffb8  eip f0100069  args 00000004 00000005 00000000 00010094
    00010094
      kern/init.c:16: test_backtrace+41
  ebp f010ffd8  eip f01000ea  args 00000005 00001aac 00000644 00000000
    00000000
      kern/init.c:43: i386_init+77
  ebp f010fff8  eip f010003e  args 00111021 00000000 00000000 00000000
    00000000
      kern/entry.S:83: <unknown>+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> help
help - Display this list of commands
kerninfo - Display information about the kernel
backtrace - Display information about the stack backtrace
K> backtrace
Stack backtrace:
  ebp f010ff68  eip f0100927  args 00000001 f010ff80 00000000 f010ffc8
```

```

f0112540
    kern/monitor.c:144: monitor+251
ebp f010ffd8  eip f01000f6  args 00000000 00001aac 00000644 00000000
00000000
    kern/init.c:43: i386_init+89
ebp f010fff8  eip f010003e  args 00111021 00000000 00000000 00000000
00000000
    kern/entry.S:83: <unknown>+0
K>

```

注意到，文件GNUmakefile中，使用的GCC命令加入了-O1优化选项，优化会导致经常调用的函数被内联到其他函数中，以减少函数调用的代价。



```

终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
NM      := $(GCCPREFIX)nm

# Native commands
NCC     := gcc-4.7 $(CC_VER) -pipe
NATIVE_CFLAGS := $(CFLAGS) $(DEFS) $(LABDEFS) -I$(TOP) -MD -Wall
TAR     := gtar
PERL    := perl

# Compiler flags
# -fno-builtin is required to avoid refs to undefined functions in the kernel.
# Only optimize to -O1 to discourage inlining, which complicates backtraces.
CFLAGS := $(CFLAGS) $(DEFS) $(LABDEFS) -O1 -fno-builtin -I$(TOP) -MD
CFLAGS += -fno-omit-frame-pointer
CFLAGS += -Wall -Wno-format -Wno-unused -Werror -gstabs -m32
# -fno-tree-ch prevented gcc from sometimes reordering read_ebp() before
# mon_backtrace()'s function prologue on gcc version: (Debian 4.7.2-5) 4.7.2
CFLAGS += -fno-tree-ch

# Add -fno-stack-protector if the option exists.
CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 &
& echo -fno-stack-protector)

# Common linker flags
~/oslab/lab1/src/lab1_1/GNUmakefile" 293L, 8529C      86,43      27%

```

Figure 31: GNUmakefile

如果去掉这个优化选项，则在内核中输入命令backtrace会显示更多的函数调用细节（调用了kern/monitor.c中的runcmd函数）：

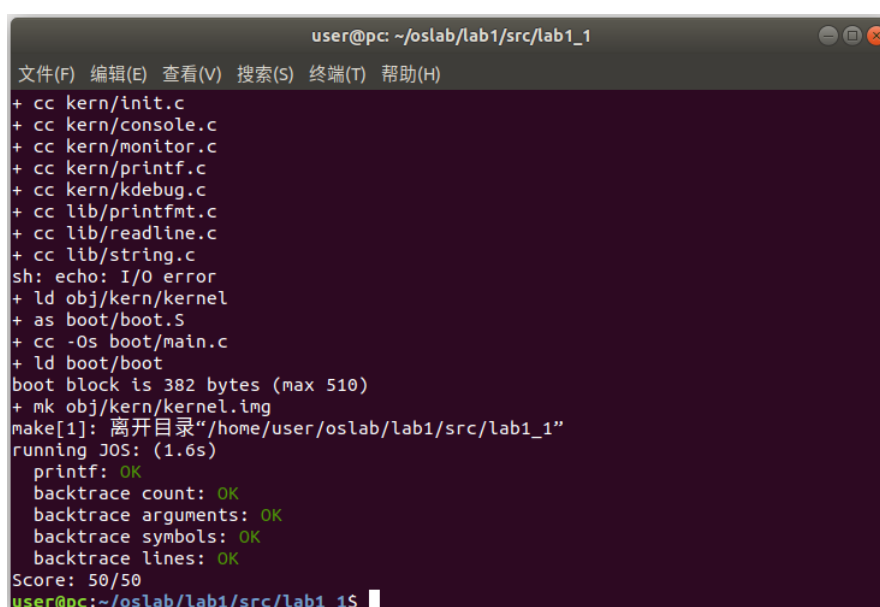
```

K> backtrace
Stack backtrace:
ebp f0110f38  eip f0100da4  args 00000001 f0110f58 00000000 0000000a
f01028bc
    kern/monitor.c:126: runcmd+302
ebp f0110fa8  eip f0100e14  args f01137a9 00000000 f0110fd8 f010009e
f010235c
    kern/monitor.c:144: monitor+72

```

```
ebp f0110fd8  eip f01000ff  args 00000000 00001aac 00000660 00000000
00000000
    kern/init.c:43: i386_init+95
ebp f0110ff8  eip f010003e  args 00000003 00001003 00002003 00003003
00004003
    kern/entry.S:83: <unknown>+0
K>
```

至此，Lab 1 的内容已完成。以下是在终端输入命令 `make grade` 之后得到的打分情况（其实最终又把优化选项加回来了）：



```
user@pc: ~/oslab/lab1/src/lab1_1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
+ 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
sh: echo: I/O error
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: 离开目录"/home/user/oslab/lab1/src/lab1_1"
running JOS: (1.6s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
user@pc:~/oslab/lab1/src/lab1_1$
```

Figure 32: 打分情况

5 总结

虽然在刚开始拿到实验指导的时候就感觉阅读量特别大、看不懂并且无从下手，但是多读了几遍实验指导、也参考了网上的一些资料之后，就稍微有了一点点做实验的眉目。通过本次实验，不仅熟悉了实验使用的 QEMU 平台和 JOS 操作系统内核，也加深了对 PC 的启动和引导过程、内核的输入输出和堆栈等知识的理解，收获还是不少的。但是在操作系统理论知识的学习上仍然存在很多知识漏洞，需要在课余时间查漏补缺。希望我们能够在后续实验有更多的收获。