

实验项目：UCore 启动实验

姓名: 张伟焜 学号: 17343155 邮箱: zhangwk8@mail2.sysu.edu.cn

院系：数据科学与计算机学院 专业：17 级软件工程 指导教师：张永东

【实验题目】

UCore 启动实验

【实验目的】

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader 来完成这些工作。为此，我们需要完成一个能够切换到 x86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。lab1 提供了一个非常小的 bootloader 和 ucoreOS，整个 bootloader 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个 bootloader 和 ucore OS，实验者可以了解到：

- 基于分段机制的存储管理
- 设备管理的基本概念
- PC 启动 bootloader 的过程
- bootloader 的文件组成
- 编译运行 bootloader 的过程
- 调试 bootloader 的方法
- ucore OS 的启动过程
- 在汇编级了解栈的结构和处理过程
- 中断处理机制
- 通过串口/并口/CGA 输出字符的方法

【实验要求】

根据指导, 完成练习 1~6。

【实验方案】

实验环境：老师提供的虚拟机（Virtual box），无特殊硬件要求

实验思路：根据实验指导，先了解理论知识，再进行实验

【实验过程】

练习 1: 理解通过 make 生成执行文件的过程。

1.

查看 Makefile 代码:

```

176# -----
177
178# create ucore.img
179UCOREIMG          := $(call totarget,ucore.img)
180
181$(UCOREIMG): $(kernel) $(bootblock)
182    $(V)dd if=/dev/zero of=$@ count=10000
183    $(V)dd if=$(bootblock) of=$@ conv=notrunc
184    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
185
186$(call create_target,ucore.img)
187
188# -----

```

运行 Makefile:

```
zhangweikun$>cd lab1
zhangweikun$>ls
boot  kern  libs  Makefile  tools
zhangweikun$>make "V="
```

显示结果及相关注释:

```
+ cc kern/init/init.c          //编译 init.c
    gcc -c kern/init/init.c -o obj/kern/init/init.o

+ cc kern/libs/readline.c      //编译 readline.c
    gcc -c kern/libs/readline.c -o
    obj/kern/libs/readline.o

+ cc kern/libs/stdio.c         //编译 stdio.c
    gcc -c kern/libs/stdio.c -o obj/kern/libs/stdio.o

+ cc kern/debug/kdebug.c       //编译 kdebug.c
    gcc -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o

+ cc kern/debug/kmonitor.c     //编译 kmonitor.c
    gcc -c kern/debug/kmonitor.c -o
    obj/kern/debug/kmonitor.o

+ cc kern/debug/panic.c        //编译 panic.c
    gcc -c kern/debug/panic.c -o obj/kern/debug/panic.o

+ cc kern/driver/clock.c       //编译 clock.c
    gcc -c kern/driver/clock.c -o obj/kern/driver/clock.o

+ cc kern/driver/console.c     //编译 console.c
    gcc -c kern/driver/console.c -o
    obj/kern/driver/console.o

+ cc kern/driver/intr.c        //编译 intr.c
    gcc -c kern/driver/intr.c -o obj/kern/driver/intr.o

+ cc kern/driver/picirq.c      //编译 picirq.c
    gcc -c kern/driver/picirq.c -o
    obj/kern/driver/picirq.o

+ cc kern/trap/trap.c          //编译 trap.c
    gcc -c kern/trap/trap.c -o obj/kern/trap/trap.o

+ cc kern/trap/trapentry.S     //编译 trapentry.S
    gcc -c kern/trap/trapentry.S -o
```

```

obj/kern/trap/trapentry.o

+ cc kern/trap/vectors.S          //编译 vectors.S
gcc -c kern/trap/vectors.S -o obj/kern/trap/vectors.o

+ cc kern/mm/pmm.c                //编译 pmm.c
gcc -c kern/mm/pmm.c -o obj/kern/mm/pmm.o

+ cc libs/printfmt.c              //编译 printfmt.c
gcc -c libs/printfmt.c -o obj/libs/printfmt.o

+ cc libs/string.c                //编译 string.c
gcc -c libs/string.c -o obj/libs/string.o

+ ld bin/kernel                    //链接成 kernel
ld -o bin/kernel
obj/kern/init/init.o      obj/kern/libs/readline.o
obj/kern/libs/stdio.o     obj/kern/debug/kdebug.o
obj/kern/debug/kmonitor.o obj/kern/debug/panic.o
obj/kern/driver/clock.o   obj/kern/driver/console.o
obj/kern/driver/intr.o    obj/kern/driver/picirq.o
obj/kern/trap/trap.o      obj/kern/trap/trapentry.o
obj/kern/trap/vectors.o   obj/kern/mm/pmm.o
obj/libs/printfmt.o       obj/libs/string.o

+ cc boot/bootasm.S              //编译 bootasm.c
gcc -c boot/bootasm.S -o obj/boot/bootasm.o

+ cc boot/bootmain.c             //编译 bootmain.c
gcc -c boot/bootmain.c -o obj/boot/bootmain.o

+ cc tools/sign.c                 //编译 sign.c
gcc -c tools/sign.c -o obj/sign/tools/sign.o
gcc -O2 obj/sign/tools/sign.o -o bin/sign

+ ld bin/bootblock                //根据 sign 规范生成 bootblock
ld -m elf_i386 -nostdlib -N -e start -Text 0x7C00
obj/boot/bootasm.o obj/boot/bootmain.o
-o obj/bootblock.o

//创建大小为 10000 个块的 ucore.img，初始化为 0，每个块为 512 字节
dd if=/dev/zero of=bin/ucore.img count=10000
//把 bootblock 中的内容写到第一个块
dd if=bin/bootblock of=bin/ucore.img conv=notrunc

```

```
//从第二个块开始写 kernel 中的内容
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
```

我们可以得到 ucore.img 的生成过程:

(1)编译所有生成 bin/kernel 所需的文件

(2)链接生成 bin/kernel

(3)编译 bootasm.S bootmain.c sign.c

(4)根据 sign 规范生成 obj/bootblock.o

(5)生成 ucore.img (首先先创建一个大小为 10000 字节的块儿, 然后再将 bootblock 拷贝过去。生成 ucore.img 需要先生成 kernel 和 bootblock)

2.文档中写道“sign.c 是一个 C 语言小程序, 是辅助工具, 用于生成一个符合规范的硬盘主引导扇区。”因此, 阅读 sign.c 代码:

```
22 char buf[512];
23 memset(buf, 0, sizeof(buf));

31 buf[510] = 0x55;
32 buf[511] = 0xAA;
```

得出, 一个被系统认为是符合规范的硬盘主引导扇区的特征是:
大小为 512 字节; 第 510 个字节为 0x55; 第 511 字节为 0xAA

练习 2: 使用 qemu 执行并调试 lab1 中的软件。

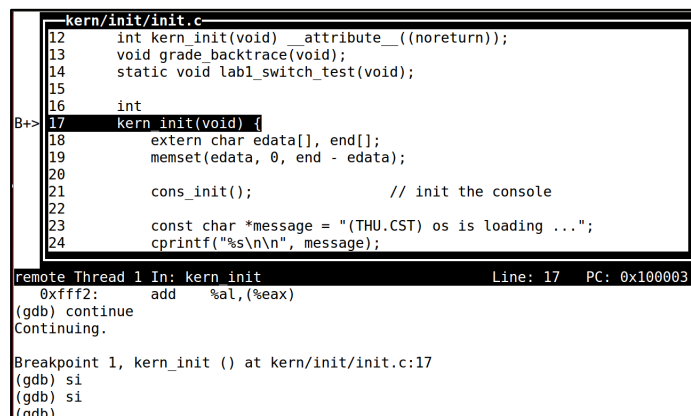
1.查看 gdbinit 文件中的指令, 它在 kern_init 函数设了断点, 再执行了 continue。我们要将 continue 去掉, 这样 make debug 后就会自动停在 0xffff0。

```
zhangweikun$>make debug
```

跳转至 gdb 界面可发现, 此时停在断点 0xffff0 (图中红色标记)



输入 continue 输入 si(stepi)进行单步调试:



2.在初始化位置 0x7c00 设置实地址断点。

输入 b*0x7c00

输入 continue 运行至断点处

```
kern/init/init.c

[ No Source Available ]

Remote Thread 1 In: Line: ?? PC: 0x7c00
Breakpoint 1, kern_init () at kern/init/init.c:17
(gdb) b*0x7c00
Breakpoint 2 at 0x7c00
(gdb) continue
Continuing.
Breakpoint 2, 0x00007c00 in ?? ()
(gdb)
```

输入 x /2i \$pc 查看反汇编代码：

```
(gdb) x /2i $pc
=> 0x7c00:      cli
      0x7c01:      cld
(gdb)
```

与 bootasm.S 中代码一致：

```
12# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
13.globl start
14start:
15.code16                                     # Assemble for 16-bit mode
16      cli                                 # Disable interrupts
17      cld                                 # String operations increment
18
```

与 bootblock.asm 中代码一致：

```
1|
2obj/bootblock.o:      file format elf32-i386
3
4
5Disassembly of section .text:
6
700007c00 <start>:
8
9# start address should be 0:7c00, in real mode, the beginning address of the running
  bootloader
10.globl start
11start:
12.code16                                     # Assemble for 16-bit mode
13      cli                                 # Disable interrupts
14      7c00:      fa                cli
15      cld                                 # String operations increment
16      7c01:      fc                cld
```

3.将 Makefile 的 debug 改为如下内容，再次 make debug 会在 bin 目录下生成 q.log

```
219debug: $(UCOREIMG)
220      $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -parallel
  stdio -hda $< -serial null"
221      $(V)sleep 2
222      $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
```



对比 q.log、bootasm.S 和 bootblock.asm 中代码，发现它们相同 (q.log 中断点后的代码与其余两者相同)：

q.log:

```
q.log x
40013
40614 -----
40615 IN:
40616 0x00007c00: cli
40617 -----
40618 -----
40619 IN:
40620 0x00007c01: cld
40621 0x00007c02: xor    %ax,%ax
40622 0x00007c04: mov    %ax,%ds
40623 0x00007c06: mov    %ax,%es
40624 0x00007c08: mov    %ax,%ss
40625 -----
40626 -----
40627 IN:
40628 0x00007c0a: in     $0x64,%al
40629 -----
40630 -----
40631 IN:
40632 0x00007c0c: test   $0x2,%al
40633 0x00007c0e: jne    0x7c0a
40634 -----
40635 -----
40636 IN:
40637 0x00007c10: mov    $0xd1,%al
40638 0x00007c12: out    %al,$0x64
40639 0x00007c14: in     $0x64,%al
40640 0x00007c16: test   $0x2,%al
```

bootasm.S:

```
q.log x bootasm.S x bootblock.asm x
12# start address should be 0:7c00, in real mode, the beginning address of the running bootloader
13.globl start
14.start:
15.code16                                # Assemble for 16-bit mode
16    cli                                # Disable interrupts
17    cld                                # String operations increment
18
19    # Set up the important data segment registers (DS, ES, SS).
20    xorw %ax, %ax                      # Segment number zero
21    movw %ax, %ds                      # -> Data Segment
22    movw %ax, %es                      # -> Extra Segment
23    movw %ax, %ss                      # -> Stack Segment
24
25    # Enable A20:|
26    # For backwards compatibility with the earliest PCs, physical
27    # address line 20 is tied low, so that addresses higher than
28    # 1MB wrap around to zero by default. This code undoes this.
29.seta20.1:
30    inb $0x64, %al                    # Wait for not busy(8042 input buffer empty).
31    testb $0x2, %al
32    jnz seta20.1
33
34    movb $0xd1, %al                   # 0xd1 -> port 0x64
35    outb %al, $0x64                  # 0xd1 means: write data to 8042's P2 port
36
37.seta20.2:
38    inb $0x64, %al                    # Wait for not busy(8042 input buffer empty).
```

bootblock.asm:

```
q.log x bootasm.S x bootblock.asm x
10.globl start
11.start:
12.code16                                # Assemble for 16-bit mode
13    cli                                # Disable interrupts
14    7c00:    fa                        cli
15    cld                                # String operations increment
16    7c01:    fc                        cld
17
18    # Set up the important data segment registers (DS, ES, SS).
19    xorw %ax, %ax                      # Segment number zero
20    7c02:    31 c0                    xor    %eax,%eax
21    movw %ax, %ds                      # -> Data Segment
22    7c04:    8e d8                    mov    %eax,%ds
23    movw %ax, %es                      # -> Extra Segment
24    7c06:    8e c0                    mov    %eax,%es
25    movw %ax, %ss                      # -> Stack Segment
26    7c08:    8e d0                    mov    %eax,%ss
27
28.00007c0a <seta20.1>:
29    # Enable A20:|
30    # For backwards compatibility with the earliest PCs, physical
31    # address line 20 is tied low, so that addresses higher than
32    # 1MB wrap around to zero by default. This code undoes this.
33.seta20.1:
34    inb $0x64, %al                    # Wait for not busy(8042 input buffer empty).
35    7c0a:    e4 64                    in     $0x64,%al
36    testb $0x2, %al
37    7c0c:    a8 02                    test   $0x2,%al
```

练习 3：分析 bootloader 进入保护模式的过程。

1. 首先关闭中断，将各个段寄存器重置为 0：

```
19    # Set up the important data segment registers (DS, ES, SS).
20    xorw %ax, %ax                # Segment number zero
21    movw %ax, %ds                # -> Data Segment
22    movw %ax, %es                # -> Extra Segment
23    movw %ax, %ss                # -> Stack Segment
```

2. 开启 A20. (当 A20 地址线控制禁止时，处于实模式。通过修改 A20 地址线可以完成从实模式到保护模式的转换，即需要通过将键盘控制器上的 A20 线置于高电位，使得全部 32 条地址线可用)

```
25    # Enable A20:
26    # For backwards compatibility with the earliest PCs, physical
27    # address line 20 is tied low, so that addresses higher than
28    # 1MB wrap around to zero by default. This code undoes this.
29 seta20.1:
30    inb $0x64, %al                # Wait for not busy(8042 input buffer empty).
31    testb $0x2, %al
32    jnz seta20.1
33
34    movb $0xd1, %al                # 0xd1 -> port 0x64
35    outb %al, $0x64                # 0xd1 means: write data to 8042's P2 port
36
37 seta20.2:
38    inb $0x64, %al                # Wait for not busy(8042 input buffer empty).
39    testb $0x2, %al
40    jnz seta20.2
41
42    movb $0xdf, %al                # 0xdf -> port 0x60
43    outb %al, $0x60                # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to
1
```

其中：

```
inb $0x64, %al    # 读取状态寄存器,等待 8042 键盘控制器闲置
```

```
testb $0x2, %al    # 判断输入缓存是否为空
```

```
movb $0xd1, %al    # 0xd1 表示写输出端口命令，参数随后通过 0x60 端口写入
```

```
movb $0xdf, %al    # 通过 0x60 写入数据 11011111 即将 A20 置 1
```

3. 加载全局描述符表 (GDT 表已经存储在引导区中，直接加载即可)：

```
45    # Switch from real to protected mode, using a bootstrap GDT
46    # and segment translation that makes virtual addresses
47    # identical to physical addresses, so that the
48    # effective memory map does not change during the switch.
49    lgdt gdt_desc
50    movl %cr0, %eax
51    orl $CR0_PE_ON, %eax
52    movl %eax, %cr0
```

4. 将控制寄存器 CR0 的第 0 位置为 1

```
50    movl %cr0, %eax
51    orl $CR0_PE_ON, %eax
52    movl %eax, %cr0
```

5. 长跳转到 32 位代码段，重装 CS 和 EIP

```
54    # Jump to next instruction, but in 32-bit code segment.
55    # Switches processor into 32-bit mode.
56    ljmp $PROT_MODE_CSEG, $protcseg
```

6. 重装 DS、ES 等段寄存器，建立堆栈

```

58 .code32                                     # Assemble for 32-bit mode
59 protseg:
60     # Set up the protected-mode data segment registers
61     movw $PROT_MODE_DSEG, %ax               # Our data segment selector
62     movw %ax, %ds                           # -> DS: Data Segment
63     movw %ax, %es                           # -> ES: Extra Segment
64     movw %ax, %fs                           # -> FS
65     movw %ax, %gs                           # -> GS
66     movw %ax, %ss                           # -> SS: Stack Segment

```

7. 转到保护模式完成，进入 bootmain 方法

```

68     # Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
69     movl $0x0, %ebp
70     movl $start, %esp
71     call bootmain

```

练习 4：分析 bootloader 加载 ELF 格式的 OS 的过程。

1. bootloader 读取硬盘扇区。

由 bootmain 函数分析，首先是由 readseg 函数读取硬盘扇区，而 readseg 函数则循环调用了真正读取硬盘扇区的函数 readsect 来一次读取一个扇区。

readseg 函数：

```

63 /* *
64 * readseg - read @count bytes at @offset from kernel into virtual address @va,
65 * might copy more than asked.
66 * */
67 static void
68 readseg(uintptr_t va, uint32_t count, uint32_t offset) {
69     uintptr_t end_va = va + count;
70
71     // round down to sector boundary
72     va -= offset % SECTSIZE;
73
74     // translate from bytes to sectors; kernel starts at sector 1
75     uint32_t secno = (offset / SECTSIZE) + 1;
76
77     // If this is too slow, we could read lots of sectors at a time.
78     // We'd write more to memory than asked, but it doesn't matter --
79     // we load in increasing order.
80     for (; va < end_va; va += SECTSIZE, secno++) {
81         readsect((void *)va, secno);
82     }
83 }

```

Readsect 函数：

```

43 /* readsect - read a single sector at @secno into @dst */
44 static void
45 readsect(void *dst, uint32_t secno) {
46     // wait for disk to be ready
47     waitdisk();
48
49     outb(0x1F2, 1); // count = 1
50     outb(0x1F3, secno & 0xFF);
51     outb(0x1F4, (secno >> 8) & 0xFF);
52     outb(0x1F5, (secno >> 16) & 0xFF);
53     outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
54     outb(0x1F7, 0x20); // cmd 0x20 - read sectors
55
56     // wait for disk to be ready
57     waitdisk();
58
59     // read a sector
60     insl(0x1F0, dst, SECTSIZE / 4);
61 }

```


其中：

```
waitdisk(); // 等待硬盘就绪
```

```
outb(...)
```

```
...// 写地址 0x1f2~0x1f5,0x1f7,发出读取磁盘的命令
```

```
insl(0x1F0, dst, SECTSIZE / 4); // 读取一个扇区
```

2.bootloader 加载 ELF 格式的 OS。

首先判断是不是 ELF 格式的文件。若不是，goto bad。

```
91 // is this a valid ELF?
92 if (ELFHDR->e_magic != ELF_MAGIC) {
93     goto bad;
94 }
95
96 struct proghdr *ph, *eph;
```

ELF 头部有描述 ELF 文件应加载到内存什么位置的描述表，这里读取出来将之存入 ph，并按照程序头表的描述，将 ELF 文件中的数据载入内存。

```
98 // load each program segment (ignores ph flags)
99 ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
100 eph = ph + ELFHDR->e_phnum;
101 for (; ph < eph; ph++) {
102     readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
103 }
```

根据 ELF 头表中的入口信息，找到内核的入口并开始运行

```
105 // call the entry point from the ELF header
106 // note: does not return
107 ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();
```

练习 5：实现函数调用堆栈跟踪函数。

思路：

先通过 read_ebp() 和 read_eip() 函数来获取当前 ebp 寄存器以及 eip 寄存器信息。然后通过 ebp+12, ebp+16, ebp+20, ebp+24 来输出 4 个参数的值，最后更新 ebp: ebp=ebp[0], 更新 eip: eip=ebp[1]。直到 ebp 对应地址的值为 0（表示当前函数为 bootmain）。

根据提供的注释编写代码：

```
292 print_stackframe(void) {
293     /* LAB1 YOUR CODE : STEP 1 */
294     /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
295      * (2) call read_eip() to get the value of eip. the type is (uint32_t);
296      * (3) from 0 .. STACKFRAME_DEPTH
297      *     (3.1) printf value of ebp, eip
298      *     (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp +2 [0..4]
299      *     (3.3) printf("\n");
300      *     (3.4) call print_debuginfo(eip-1) to print the C calling function name and line number, etc.
301      *     (3.5) popup a calling stackframe
302      *         NOTICE: the calling function's return addr eip = ss:[ebp+4]
303      *         the calling function's ebp = ss:[ebp]
304      */
305     uint32_t ebp=read_ebp(); // (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
306     uint32_t eip=read_eip(); // (2) call read_eip() to get the value of eip. the type is (uint32_t);
307     int i;
308     for(i=0; i<STACKFRAME_DEPTH&&ebp!=0; i++){ // (3) from 0 .. STACKFRAME_DEPTH
309         printf("ebp:0x%08x eip:0x%08x ", ebp, eip); // (3.1) printf value of ebp, eip
310         uint32_t *temp=(uint32_t *)ebp+2;
311         printf("args:0x%08x 0x%08x 0x%08x 0x%08x", *(temp+0), *(temp+1), *(temp+2), *(temp+3)); //
312         (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp +2 [0..4]
313         printf("\n"); // (3.3) printf("\n");
314         print_debuginfo(eip-1); // (3.4) call print_debuginfo(eip-1) to print the C calling
315         function name and line number, etc.
316         eip=((uint32_t *)ebp)[1];
317         ebp=((uint32_t *)ebp)[0]; // (3.5) popup a calling stackframe
318     }
```

运行结果:

```
QEMU
Special kernel symbols:
  entry 0x00100000 (phys)
  etext 0x001032cd (phys)
  edata 0x0010ea16 (phys)
  end 0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a7 args:0x00000000 0x00007b38 0x00100092
  kern/debug/kdebug.c:306: print_stackframe+22
ebp:0x00007b10 eip:0x00100c9f args:0x00000000 0x00000000 0x00007b08
  kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
  kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
  kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
  kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032fc 0x001032e0 0x0000130a 0x00000000
  kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00010094
  kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
  <unknown>: -- 0x00007d67 --
++ setup timer interrupts
  kern/debug/kdebug.c:306: print_stackframe+22
ebp:0x00007b18 eip:0x00100c9f args:0x00000000 0x00000000 0x00000000 0x00007b88
  kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
  kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
  kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
  kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032fc 0x001032e0 0x0000130a 0x00000000
  kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00010094
  kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
  <unknown>: -- 0x00007d67 --
++ setup timer interrupts
```

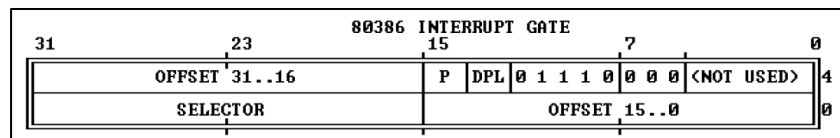
最后一行的解释:

最后一行对应的是第一个使用堆栈的函数, bootmain.c 中的 bootmain。(因为此时 ebp 对应地址的值为 0) bootloader 设置的堆栈从 0x7c00 开始, 使用"call bootmain"转入 bootmain 函数。call 指令压栈, 所以 bootmain 中 ebp 为 0x7bf8。

```
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
  <unknown>: -- 0x00007d67 --
++ setup timer interrupts
```

练习 6: 完善中断初始化和处理。

1.中断向量表中一个表项占 8 字节。其中 0~15 位和 48~63 位分别为 offset 的低 16 位和高 16 位。16~31 位为段选择子。通过段选择子获得段基址, 加上段内偏移量即可得到中断处理代码入口。



2.首先, 声明 `_vectors[]`, 其中存放着中断服务程序的入口地址。这个数组生成于 `vector.S` 中。接着, 填充中断描述符表 IDT。然后, 加载中断描述符表。

```
49 extern uintptr_t _vectors[]; // declare _vectors[]
50 int i;
51 for (i = 0; i < 256; i++) {
52     SETGATE(idt[i], 0, GD_KTEXT, _vectors[i], DPL_KERNEL);
53 }
54 // set for switch from user to kernel
55 SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, _vectors[T_SWITCH_TOK], DPL_USER);
56 // load the IDT
57 lidt(&idt_pd);
```

其中 $i < 256$, 是参考代码中给的注释:

```
43      * (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).
44      *      Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to
      setup each item of IDT
```

另外, SETGATE 的定义在 mmu.h 中:

```
71 #define SETGATE(gate, istrap, sel, off, dpl) {
72     (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;
73     (gate).gd_ss = (sel);
74     (gate).gd_args = 0;
75     (gate).gd_rsv1 = 0;
76     (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;
77     (gate).gd_s = 0;
78     (gate).gd_dpl = (dpl);
79     (gate).gd_p = 1;
80     (gate).gd_off_31_16 = (uint32_t)(off) >> 16;
81 }
```

gate: 对应的 idt[] 数组内容, 处理函数的入口地址

istrap: 系统段设置为 1, 中断门设置为 0

sel: 段选择子

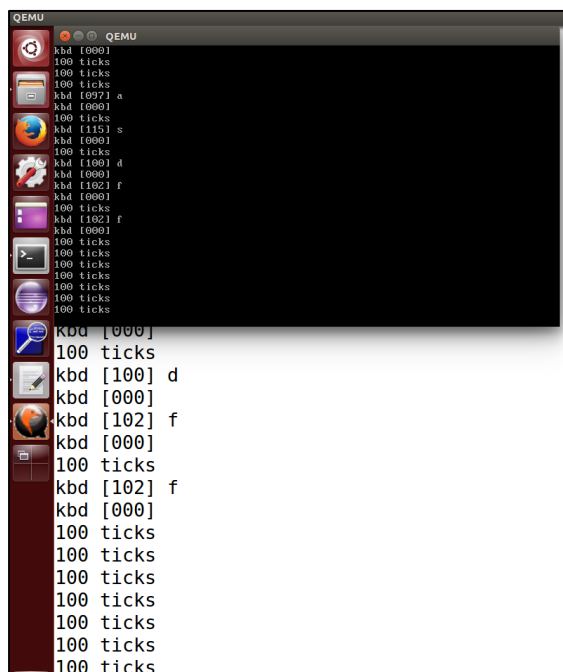
off: __vectors[] 数组内容

dpl: 设置特权级。这里中断都设置为内核级, 即第 0 级

3. trap 函数调用子函数 trap_dispatch, 补全 trap_dispatch 函数的判断部分:

```
152     case IRQ_OFFSET + IRQ_TIMER:
153         /* LAB1 YOUR CODE : STEP 3 */
154         /* handle the timer interrupt */
155         /* (1) After a timer interrupt, you should record this event using a global variable
          (increase it), such as ticks in kern/driver/clock.c
156         * (2) Every TICK_NUM cycle, you can print some info using a function, such as
          print_ticks().
157         * (3) Too Simple? Yes, I think so!
158         */
159         ticks++;
160         if (ticks % TICK_NUM == 0) {
161             print_ticks();
162         }
163         break;
```

结果: 每 100 次时钟中断显示“100 ticks”, 且按下的键也会在屏幕上显示。



【实验总结】

初次拿到实验文档，感觉压力很大，认为实验很难，但当自己真正静心阅读实验指导时发现其实难度没有自己想象的那么大。甚至有很多问题的答案直接就写在实验指导中，如中断向量表就直接在指导手册中画出来了。但不可否认的是，实验的内容还是挺丰富的，涉及的方面也很多，包括到 makefile, gdb 调试等基础操作，也涉及到 BIOS, 扇区引导，中断机制，函数调用堆栈等内容。

实验遇到的一个困难是，开始的时候陷在 makefile 里无法自拔，花费了很长时间。自己过于纠结每个 makefile 的语句，忽略了对整个编译先后顺序的整体把握。同时，由于对 gdb 调试不熟悉，花了一些时间去学习，但真正做实验的时候发现需要用的指令并不是很多。

另外，在实验过程中学习了一些新操作，比如 makefile 的 `make V=` 帮助我们理解编译过程；在 gdb 中用 `x/2i %pc` 进行反汇编查看汇编代码。

总之，通过这次试验，我对 PC 启动 bootloader 以及 ucore 操作系统的启动过程有了深入的认识，掌握了中断处理机制和函数调用堆栈跟踪函数。

【参考文献】

《操作系统实验指导(清华大学)陈渝、向勇编著》