Ucore lab1 实验报告

4042017018

罗栋兰

练习 1

列出本实验各练习中对应的 OS 原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。

在此练习中，大家需要通过静态分析代码来了解：

1. 操作系统镜像文件 ucore.img 是如何一步一步生成的？(需要比较详细地解释 Makefile 中每一条相关命令和命令参数的含义，以及说明命令导致的结果)
2. 一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

解答：

1.1 利用 make V= 查看执行了那些命令

首先我们查看 lab1 下的 Makefile 文件
找到 create ucrore.image 大概在 169h

```
$(UCOREIMG): $(kernel) $(bootblock)

    $(V)dd if=/dev/zero of=$@ count=10000

    $(V)dd if=$(bootblock) of=$@ conv=notrunc

    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

    $(call create_target,ucore.img)
```

由上描述可以看出，首先先创建一个大小为 10000 字节的块，然后再将 bootblock，kernel 拷贝过去。然而生成 ucore.img 需要先生成 kernel 和 bootblock

dd 是一条 linux 命令指令：

dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换。

if=文件名：输入文件名，缺省为标准输入。即指定源文件。< if=input file >

of=文件名：输出文件名，缺省为标准输出。即指定目的文件。< of=output file >

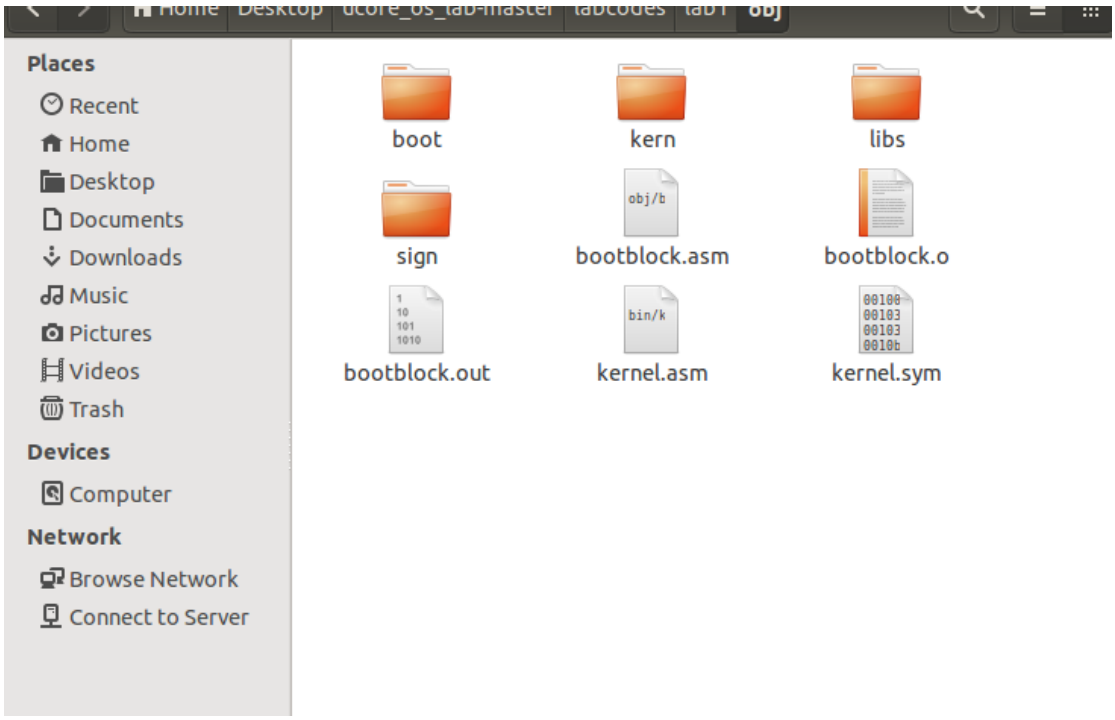
count=blocks：仅拷贝 blocks 个块，块大小等于 ibs 指定的字节数。
conv=conversion：用指定的参数转换文件。 conv=notrunc:不截短输出文件

生成 bootblock 的相关代码如下

```
$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)

	@echo "==========================$(call toobj,$(bootfiles))"

	@echo + ld $@

	$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)

	@$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)

	@$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)

	@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
```

由上代码可得，到要生成 bootblock，首先需要生成 bootasm.o、bootmain.o、sign

下图是在编译时生成的中间文件



拷贝二进制代码 bootblock.o 到 bootblock.out

objcopy -S -O binary obj/bootblock.o obj/bootblock.out

其中关键的参数为

-S 移除所有符号和重定位信息

-O 指定输出格式

使用 sign 工具处理 bootblock.out，生成 bootblock

bin/sign obj/bootblock.out bin/bootblock

kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)

    @echo + ld $@

    (V)(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)

    @$(OBJDUMP) -S $@ > $(call asmfile,kernel)

```
	@$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /;
/^$$/d' > $(call symfile,kernel)

$(call create_target,kernel
```

在终端中执行输入 make V= 会看到 ucore.image 的编译内容
正是我们之前说的先创建一个大小为 10000 字节的内存块儿，然后再将 bootblock 和 kernel 拷贝过去

1.2 查看 sign.c 源代码

```
buf[510] = 0x55;

buf[511] = 0xAA;

FILE *ofp = fopen(argv[2], "wb+");

size = fwrite(buf, 1, 512, ofp);

if (size != 512) {

	fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);

	return -1;

}
```

从上述代码可以看出，要求硬盘主引导扇区的大小是 512 字节，还需要第 510 个字节是 0x55,第 511 个字节为 0xAA,也就是说扇区的最后两个字节内容是 0x55AA

练习 2

bios 是储存在固件当中的一段代码。在操作系统启动之前对硬件进行检查，然后再加载运行引导区的代码。

2.1.从 CPU 加电后执行的第一条指令开始，单步跟踪 BIOS 的执行。
其实就是用 qemu 和 gdb 进行调试，需要了解一些简单的 gdb 命令

1. lab1/tools/gdbinit 文件为

set architecture i8086

target remote:1234

意思是与 qemu 建立连接

2. 执行以下命令

Make debug

3. 4.gdb 界面下用 si 命令单步跟踪，可通过如下命令来看 BIOS 的代码

x /2i $pc

2.2.在初始化位置 0x7c00 设置实地址断点,测试断点正常。
在初始化位置 0x7c00 设置实地址断点，测试断点正常。

修改 gdbinit 文件：

```
set architecture i8086
target remote :1234
b *0x7c00
c
x/2i $pc
```

由于我们需要观察电脑在 0x7c00 处电脑的运行情况，所以首先需要在地址为 0x7c00 的地方设置断点，使得电脑在此停住：然后通过 continue 指令，使得程序继续运行，直到运行到 0x7c00 之后再次停住。再次通过 x/10i $pc 查看相近的指令：通过查看 bootasm.S 文件我们可以看到，此时电脑已经进入 bootasm.S 文件，开始执行相应的代码。然后再次输入 c 我们看到 qemu 工作正常，所以断点正常

2.3.从 0x7c00 开始跟踪代码运行,将单步跟踪反汇编得到的代码与 bootasm.S 和 bootblock.asm 进行比较

改写 makefile 文件：

debug: $(UCOREIMG)

	(V)(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -parallel stdio -hda $< -serial null"

	$(V)sleep 2

	(V)(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"

然后再执行

make debug

得到 q.log 文件：

```
----------------
IN:
0x00007c00:  cli
0x00007c01:  cld
0x00007c02:  xor    %ax,%ax
0x00007c04:  mov    %ax,%ds
0x00007c06:  mov    %ax,%es
```

bootasm.S 文件中的代码和 bootblock.asm 是一样的，对于 q.log 文件，断点之后的代码和 bootasm.S，bootblock.asm 是一样的

自己找一个 bootloader 或内核中的代码位置，设置断点并进行测试。

修改 gdbinit 文件，在 0x7c4a 处设置断点 (调用 bootmain 函数处)

set architecture i8086

target remote :1234

break *0x7c4a

```
---------------
IN:
0x00007c00:  cli
0x00007c01:  cld
0x00007c02:  xor    %ax,%ax
0x00007c04:  mov    %ax,%ds
0x00007c06:  mov    %ax,%es
```

输入 make debug ,得到结果：

```
(gdb) c
Continuing.

Breakpoint 1, 0x00007c4a in ?? ()
(gdb) x/2i $pc
=> 0x7c4a:      call   0x7ccf
   0x7c4d:      add    %al,(%bx,%si)
(gdb)
```

练习 3

bootloader 中从实模式进到保护模式的代码保存在 lab1/boot/bootasm.S 文件下，使用 x86 汇编语言编写，接下来将根据源码分析进入保护模式的过程：

1 关中断和清除数据段寄存器

```
.globl start
start:
.code16
      cli                    //关中断
      cld                    //清除方向标志
      xorw %ax, %ax          //ax 清 0
      movw %ax, %ds          //ds 清 0
      movw %ax, %es          //es 清 0
      movw %ax, %ss          //ss 清 0
```

3.1 为何开启 A20，以及如何开启 A20？

初始时 A20 为 0，访问超过 1MB 的地址时，就会从 0 循环计数，将 A20 地址线置为 1 之后，才可以访问 4G 内存。A20 地址位由 8042 控制，8042 有 2 个有两个 I/O 端口：0x60 和 0x64。

打开流程：

等待 8042 Input buffer 为空；
发送 Write 8042 Output Port （P2）命令到 8042 Input buffer；
等待 8042 Input buffer 为空；
将 8042 Output Port（P2）得到字节的第 2 位置 1，然后写入 8042 Input buffer；

```
seta20.1:                    //等待 8042 键盘控制器不忙
```

```
        inb $0x64, %al      //从 0x64 端口中读入一个字节到 al 中
        testb $0x2, %al     //测试 al 的第 2 位
        jnz seta20.1        //al 的第 2 位为 0，则跳出循环

        movb $0xd1, %al     //将 0xd1 写入 al 中
        outb %al, $0x64     //将 0xd1 写入到 0x64 端口中

seta20.2:                   //等待 8042 键盘控制器不忙
        inb $0x64, %al      //从 0x64 端口中读入一个字节到 al 中
        testb $0x2, %al     //测试 al 的第 2 位
        jnz seta20.2        //al 的第 2 位为 0，则跳出循环

        movb $0xdf, %al     //将 0xdf 入 al 中
        outb %al, $0x60     //将 0xdf 入到 0x64 端口中，打开 A20
```

3.2 如何初始化 GDT 表？

1 载入 GDT 表

```
 lgdt gdtdesc           //载入 GDT 表
```

2 进入保护模式：

通过将 cr0 寄存器 PE 位置 1 便开启了保护模式

cro 的第 0 位为 1 表示处于保护模式

```
movl %cr0, %eax           //加载 cro 到 eax
orl $CR0_PE_ON, %eax    //将 eax 的第 0 位置为 1
movl %eax, %cr0           //将 cr0 的第 0 位置为 1
```

3 通过长跳转更新 cs 的基地址:

上面已经打开了保护模式，所以这里需要用到逻辑地址。$PROT_MODE_CSEG 的值为 0x80

```
ljmp $PROT_MODE_CSEG, $protcseg
.code32
protcseg:
```

4 设置段寄存器，并建立堆栈

```
 movw $PROT_MODE_DSEG, %ax //
 movw %ax, %ds
 movw %ax, %es
 movw %ax, %fs
 movw %ax, %gs
 movw %ax, %ss
```

movl $0x0, %ebp //设置帧指针
movl $start, %esp //设置栈指针
5 转到保护模式完成，进入 boot 主方法
call bootmain //调用 bootmain 函数

3.3 如何使能和进入保护模式

将 cr0 寄存器置 1

练习 4

在上一个联系中我们的 BootLoader 已经成功的进入了保护模式，接下来我们要做的就是从硬盘读取并运行我们的 OS。查看 bootman.c 源码

```
85  void
86  bootmain(void) {
87      // read the 1st page off disk
88      readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
89
90      // is this a valid ELF?
91      if (ELFHDR->e_magic != ELF_MAGIC) {
92          goto bad;
93      }
94
95      struct proghdr *ph, *eph;
96
97      // load each program segment (ignores ph flags)
98      ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
99      eph = ph + ELFHDR->e_phnum;
100     for (; ph < eph; ph ++) {
101         readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
102     }
103
104     // call the entry point from the ELF header
105     // note: does not return
106     ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
107
108 bad:
109     outw(0x8A00, 0x8A00);
110     outw(0x8A00, 0x8E00);
111
112     /* do nothing */
113     while (1);
```

ELF 就是一个可执行文件的意思

代码分析

- 88L 读取第一个扇区
- 91L 判断是否是一个有效的 elf

- 97L-103L 加载接下来的扇区
- 106L 利用强转为函数指针调用函数

4.1bootloader 如何读取硬盘扇区的？

大致流程如下：

（1）.等待磁盘准备好

（2）.发出读取扇区的命令

（3）.等待磁盘准备好

（4）.把磁盘扇区数据读取到指定内存

查看 readsect 函数 源码

```
41
42  /* readsect - read a single sector at @secno into @dst */
43  static void
44  readsect(void *dst, uint32_t secno) {
45      // wait for disk to be ready
46      waitdisk();
47
48      outb(0x1F2, 1);                         // count = 1
49      outb(0x1F3, secno & 0xFF);
50      outb(0x1F4, (secno >> 8) & 0xFF);
51      outb(0x1F5, (secno >> 16) & 0xFF);
52      outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
53      outb(0x1F7, 0x20);                      // cmd 0x20 - read sectors
54
55      // wait for disk to be ready
56      waitdisk();
57
58      // read a sector
59      insl(0x1F0, dst, SECTSIZE / 4);
```

```
62  /*
63   * readseg - read @count bytes at @offset from kernel into virtual address @va,
64   * might copy more than asked.
65   */
66  static void
67  readseg(uintptr_t va, uint32_t count, uint32_t offset) {
68      uintptr_t end_va = va + count;
69
70      // round down to sector boundary
71      va -= offset % SECTSIZE;
72
73      // translate from bytes to sectors; kernel starts at sector 1
74      uint32_t secno = (offset / SECTSIZE) + 1;
75
76      // If this is too slow, we could read lots of sectors at a time.
77      // We'd write more to memory than asked, but it doesn't matter --
78      // we load in increasing order.
79      for (; va < end_va; va += SECTSIZE, secno ++) {
80          readsect((void *)va, secno);
81      }
82  }
```

代码分析

- 48L 设置读取扇区的数目为 1
- 49L-52L

- 在这 4 个字节线联合构成的 32 位参数中
- 29-31 位强制设为 1
- 28 位(=0)表示访问"Disk 0"
- 0-27 位是 28 位的偏移量
- 53L 0x20 命令 读取扇区
- 59L 从 0x1F0 读取 SECTSIZE 字节数到 dst 的位置,每次读四个字节，读取 SECTSIZE/ 4 次。

  从 utb()可以看出这里是用 LBA 模式的 PIO（Program IO）方式来访问硬盘的。从磁盘 IO 地址和对应功能表可以看出，该函数一次只读取一个扇区。

4.2 bootloader 是如何加载 ELF 格式的 OS 的?

也就是 bootmain 函数内容

看上文的 bootmain 中 elfhdr、proghdr 相关的信息。

下面是关于 elfhdr 和 proghdr 的相关信息

```
void
bootmain(void) {
    //读取硬盘的第一个扇区
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 判断是否是有效的 ELF 文件
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    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
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    //按照程序头表的描述，将 ELF 文件中的数据载入内存
    for (; ph < eph; ph ++) {
```

```
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz,
ph->p_offset);
    }

    //根据 ELF 头表中的入口信息，找到内核的入口并开始运行
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    while (1);
```

具体流程：

1. 首先从硬盘中将 bin/kernel 文件的第一页内容加载到内存地址为 0x10000 的位置，目的是读取 kernel 文件的 ELF Header 信息。
2. 校验 ELF Header 的 e_magic 字段，以确保这是一个 ELF 文件
3. 读取 ELF Header 的 e_phoff 字段，得到 Program Header 表的起始地址；读取 ELF Header 的 e_phnum 字段，得到 Program Header 表的元素数目。
4. 遍历 Program Header 表中的每个元素，得到每个 Segment 在文件中的偏移、要加载到内存中的位置（虚拟地址）及 Segment 的长度等信息，并通过磁盘 I/O 进行加载
5. 加载完毕，通过 ELF Header 的 e_entry 得到内核的入口地址，并跳转到该地址开始执行内核代码

调试代码

练习 5：实现函数调用堆栈跟踪函数 （需要编程）

需要完成 kdebug.c 中函数 print_stackframe 的实现，可以通过函数 print_stackframe 来跟踪函数调用堆栈中记录的返回地址。

一．函数堆栈

主要的两点在于栈的结构和 ebp 寄存器的作用。一个函数调用动作可分解为：零到多个 PUSH 指令（用于参数入栈），一个 CALL 指令。CALL 指令内部其实还暗含了一个将返回地址（即 CALL 指令下一条指令

的地址）压栈的动作（由硬件完成）。几乎所有本地编译器都会在每个函数体之前插入类似如下的汇编指令：

pushl %ebp

movl %esp,%ebp

这样在程序执行到一个函数的实际命令前，已经有以下数据顺序入栈：参数，返回地址，ebp 寄存器。

函数调用的步骤：

1.参数入栈：将参数从右向左依次压入栈中。

2. 返回地址入栈：call 指令内部隐含的动作，将 call 的下一条指令入栈，由硬件　　完成。

3. 代码区跳转：跳转到被调用函数入口处。

4. 函数入口处前两条指令，为本地编译器自动插入的指令，即将 ebp 寄存器入栈，　　然后将栈顶指针 esp 赋值给 ebp。

相反的，函数返回的步骤为：

1.保存返回值，通常将函数返回值保存到寄存器 EAX 中。

2. 将当前的 ebp 赋给 esp。

3. 从栈中弹出一个值给 ebp。

4. 弹出返回地址，从返回地址处继续执行。

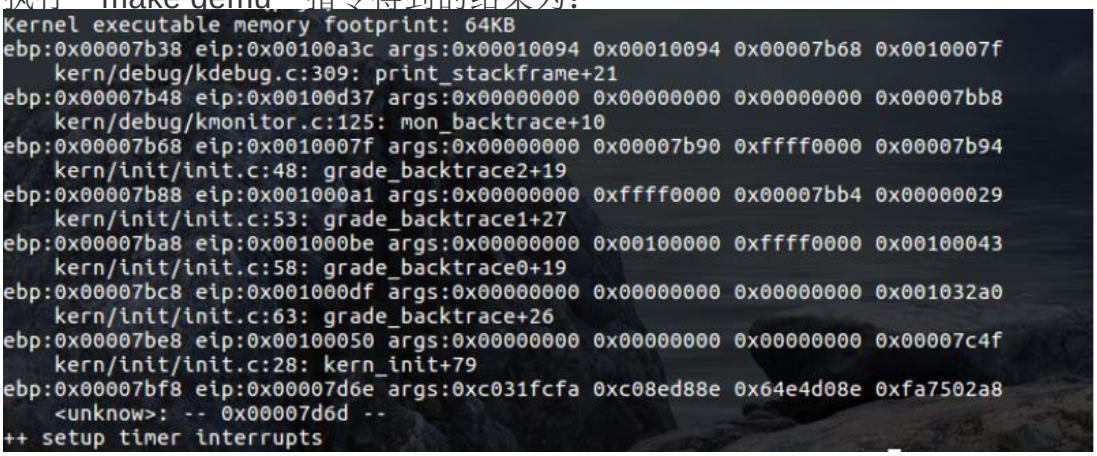
并且在函数调用过程中的 ebp 起着关键作用，从该地址向上（栈底方向）能依次获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的 ebp 的值，于是以此为线索可以形成递归，直至到达栈底。这就是函数调用栈。

二．print_stackframe 函数的实现

由以上知识和源代码文件中的注释实现 print_stackframe():

```
void
print_stackframe(void) {
        uint32_t ebp=read_ebp();    //得到当前的ebp的值
        uint32_t eip=read_eip();    //得到当前eip的值
        while(ebp){                 //循环至ebp=0,即栈底
                //打印ebp，eip的值
                cprintf("ebp:0x%08x eip:0x%08x ",(uint32_t*)ebp,(uint32_t*)eip);
                cprintf("args:");
                for(uint32_t i=0;i<4;++i)   //依次打印4个参数的值
                        cprintf("0x%08x ",*((uint32_t*)ebp+2+i));
                cprintf("\n");
                print_debuginfo(eip-1);     //查找该地址所在函数并打印函数名和行号
                eip=*((uint32_t*)ebp+1);    //更新eip为上一层函数的eip值
                ebp=*((uint32_t*)ebp);      //更新ebp为上一层函数的ebp值
```

执行’make qemu’指令得到的结果为：



以观察到显示结果与实验指导书上一致对于最后一行：其对应的是第一个调用堆栈的函数，即 bootmain.c 中的 bootmain 函数，因为 bootloader 设置的堆栈从 0x7c00 开始，执行’call bootmain’转入 bootmain 函数。其 ebp=0x00007bf8，此时的 eip=0x00007d6e，其压入的 4 个参数分别为 0xc031fcfa, 0xc08ed88e, 0x64e4d08e,

0xfa7502a8。

练习 6--完善中断初始化和处理

6.1

1. IDT 中的每一个表项均占 8 个字节；
2. 其中最开始 2 个字节和最末尾 2 个字节定义了 offset，第 16-31 位定义了处理代码入口地址的段选择子，使用其在 GDT 中查找到相应段的 base address，加上 offset 就是中断处理代码的入口。

6.2

```
void
idt_init(void) {
    extern uintptr_t __vectors[];   //保存在 vectors.S 中的 256 个中断处理例程的入口地址数组
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) { //IDT 表项的个数
      SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT,
    __vectors[T_SWITCH_TOK], DPL_USER);
    lidt(&idt_pd);
```

}

6.3

设置时钟进行操作：

```
    case IRQ_OFFSET + IRQ_TIMER:
        ticks ++;         //一次中断累加 1
          if (ticks % TICK_NUM == 0)
        {
            print_ticks();}   break;
```