

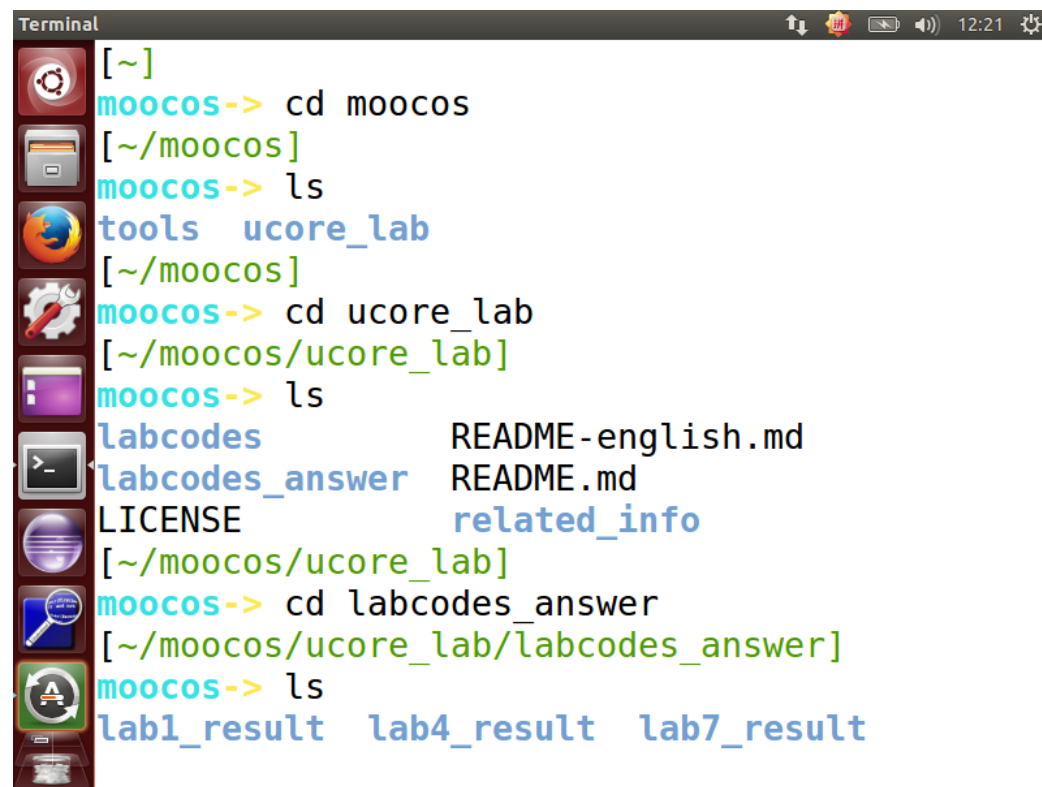
lab1

樊子晨

练习1理解通过make生成执行文件的过程

操作系统镜像文件ucore.img是如何一步一步生成的？

调试代码：



```
Terminal
[~]
moocos-> cd moocos
[~/moocos]
moocos-> ls
tools  ucore_lab
[~/moocos]
moocos-> cd ucore_lab
[~/moocos/ucore_lab]
moocos-> ls
labcodes          README-english.md
labcodes_answer   README.md
LICENSE           related_info
[~/moocos/ucore_lab]
moocos-> cd labcodes_answer
[~/moocos/ucore_lab/labcodes_answer]
moocos-> ls
lab1_result  lab4_result  lab7_result
```

```
Terminal
[~/moocos/ucore_lab]
moocos-> ls
labcodes          README-english.md
labcodes_answer   README.md
LICENSE           related_info
[~/moocos/ucore_lab]
moocos-> cd labcodes_answer
[~/moocos/ucore_lab/labcodes_answer]
moocos-> ls
lab1_result  lab4_result  lab7_result
lab2_result  lab5_result  lab8_result
lab3_result  lab6_result
[~/moocos/ucore_lab/labcodes_answer]
moocos-> cd lab1_result
[~/moocos/ucore_lab/labcodes_answer/lab1_resu
lt]
moocos->

Terminal
moocos-> cd lab1_result
[~/moocos/ucore_lab/labcodes_answer/lab1_resu
lt]
moocos-> make clean
rm -f -r obj bin
[~/moocos/ucore_lab/labcodes_answer/lab1_resu
lt]
moocos-> make V=
+ cc kern/init/init.c
gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m3
2 -gstabs -nostdinc -fno-stack-protector -Il
ibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap
/ -Ikern/mm/ -c kern/init/init.c -o obj/kern/
init/init.o
+ cc kern/libs/readline.c
gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m3
2 -gstabs -nostdinc -fno-stack-protector -Il
```

分别生成bootblock和kernel:

```
Terminal
success!
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.03046 s, 168
MB/s
dd if=bin/bootblock of=bin/ucore.img conv=not
runc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.00345339 s, 148 k
B/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv
=notrunc
146+1 records in
146+1 records out
74923 bytes (75 kB) copied, 0.000246549 s, 30
```

从实验代码可知，ld通过编译将目标文件转化成了一个执行程序——bootblock
所以 ucore.img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

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？

- 1.大小为512字节
- 2.多余的空间为0
- 3.最后两个字节是0x55AA

练习2使用qemu执行并调试lab1中的软件

利用共享文件夹将vscode列入虚拟机中同步数据

从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行

执行代码：

less Makefile

/lab1-mon

分析：

```
1 201 lab1-mon: $(UCOREIMG)
2 202 $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -mon
3 203 $(V)sleep 2
4 204 $(V)$(TERMINAL) -e "gdb -q -x tools/lab1init"
5 205 debug-mon: $(UCOREIMG)
6
```

结论：

qemu把执行的指令记录下来并放在q.log

利用gdb调试bootloader

gdb的地址断点

在gdb命令中，使用**b [地址]**便可以在指定内存地址设置断点，当qemu中的cpu执行到指定地址时，便会将控制权交给gdb。

有可能gdb无法正确获取当前qemu执行的汇编指令，通过如下配置可以在每次gdb命令行前强制反汇编当前的指令，在gdb命令行或配置文件中添加：

```
1  define hook-stop
2  x/i $pc
3  end
```

即可

在gdb中，有next, nexti, step, stepi等指令来单步调试程序，他们功能各不相同，区别在于单步的“跨度”上。

```
1  next 单步到程序源代码的下一行，不进入函数。
2  nexti 单步一条机器指令，不进入函数。
3  step 单步到下一个不同的源代码行（包括进入函数）。
4  stepi 单步一条机器指令
```

在初始化位置0x7c00设置实地址断点,测试断点正常

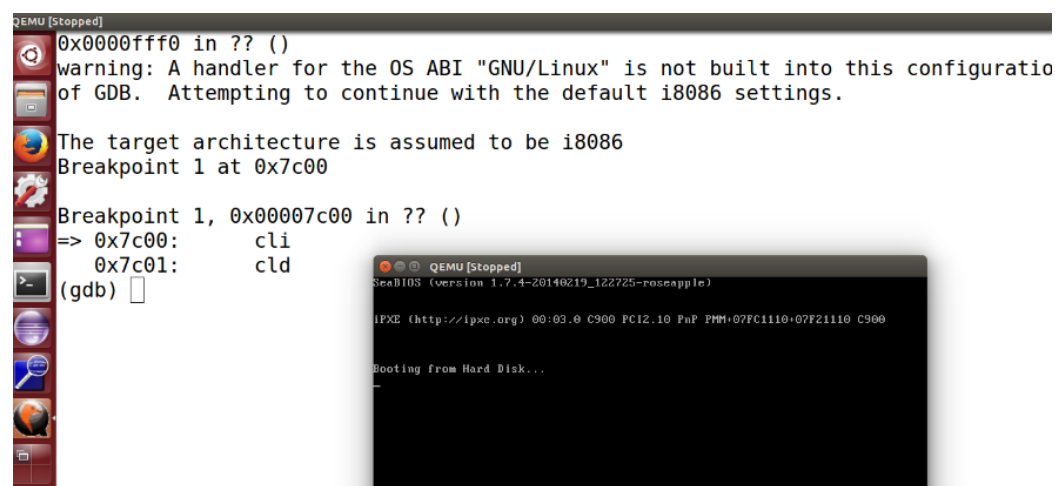
设置断点进行测试：

```
1 | less tools/lab1init
```

打开后的代码：

```
1 | 1 file bin/kernel
2 | 2 target remote :1234
3 | 3 set architecture i8086
4 | 4 b *0x7c00
5 | 5 continue
6 | 6 x /2i $pc
```

再次输入make lab1-mon



结论：qemu窗口可以启动，但在0x7c01处停止。

输入continue--继续运行

输入Ctrl+c--停止运行

练习3：分析bootloader进入保护模式的过程

BIOS将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行bootloader。

如何打开A20：

通过将键盘控制器上的A20线置于高电位，全部32条地址线可用，可以访问4G的内存空间。

```
1      seta20.1:                # 等待8042键盘控制器不忙
2          inb $0x64, %al        #
3          testb $0x2, %al       #
4          jnz seta20.1          #
5
6          movb $0xd1, %al       # 发送写8042输出端口的指令
7          outb %al, $0x64       #
8
9      seta20.1:                # 等待8042键盘控制器不忙
10         inb $0x64, %al        #
11         testb $0x2, %al       #
12         jnz seta20.1          #
13
14         movb $0xdf, %al       # 打开A20
15         outb %al, $0x60       #
16
```

如何初始化GDT表

```
1  lgdt gdt_desc                #把gdt表的起始位置和界限装入GDTR寄存器  movl %cr0, %eax
2  movl %eax, %cr0              #把保护模式位开启
```

如何使能和进入保护模式

```
1      movl %cr0, %eax
2      orl $CR0_PE_ON, %eax
3      movl %eax, %cr0
```

练习4：分析bootloader加载ELF格式的OS的过程

- bootloader如何读取硬盘扇区的？
- bootloader是如何加载ELF格式的OS？

首先看readsect函数，

readsect从设备的第secno扇区读取数据到dst位置

```

1  static void
2  readsect(void *dst, uint32_t secno) {
3      waitdisk();
4
5      outb(0x1F2, 1);                // 设置读取扇区的数目为1
6      outb(0x1F3, secno & 0xFF);
7      outb(0x1F4, (secno >> 8) & 0xFF);
8      outb(0x1F5, (secno >> 16) & 0xFF);
9      outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
10     // 上面四条指令联合制定了扇区号
11     // 在这4个字节线联合构成的32位参数中
12     //   29-31位强制设为1
13     //   28位(=0)表示访问"Disk 0"
14     //   0-27位是28位的偏移量
15     outb(0x1F7, 0x20);              // 0x20命令, 读取扇区
16
17     waitdisk();
18
19     insl(0x1F0, dst, SECTSIZE / 4); // 读取到dst位置,
20                                     // 幻数4因为这里以DW为单位
21 }

```

readseg简单包装了readsect, 可以从设备读取任意长度的内容。

```

1  static void
2  readseg(uintptr_t va, uint32_t count, uint32_t offset) {
3      uintptr_t end_va = va + count;
4
5      va -= offset % SECTSIZE;
6
7      uint32_t secno = (offset / SECTSIZE) + 1;
8      // 加1因为0扇区被引导占用
9      // ELF文件从1扇区开始
10
11     for (; va < end_va; va += SECTSIZE, secno++) {
12         readsect((void *)va, secno);
13     }
14 }

```

分析一下bootmain函数:

```

1  void
2  bootmain(void) {
3      // 首先读取ELF的头部
4      readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
5
6      // 通过储存在头部的幻数判断是否是合法的ELF文件
7      if (ELFHDR->e_magic != ELF_MAGIC) {
8          goto bad;
9      }
10
11     struct proghdr *ph, *eph;

```

```

12
13     // ELF头部有描述ELF文件应加载到内存什么位置的描述表，
14     // 先将描述表的头地址存在ph
15     ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
16     eph = ph + ELFHDR->e_phnum;
17
18     // 按照描述表将ELF文件中数据载入内存
19     for (; ph < eph; ph++) {
20         readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
21     }
22     // ELF文件0x1000位置后面的0xd1ec比特被载入内存0x00100000
23     // ELF文件0xf000位置后面的0x1d20比特被载入内存0x0010e000
24
25     // 根据ELF头部储存的入口信息，找到内核的入口
26     ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();
27
28     bad:
29         outw(0x8A00, 0x8A00);
30         outw(0x8A00, 0x8E00);
31         while (1);
32     }
33

```

由代码可以了解：

- 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

综上所述，bootloader加载ELF格式的OS的流程为

- 1 从硬盘读了8个扇区数据到内存0x10000处，并把这里强制转换成elfhdr使用
- 2 校验e_magic字段
- 3 根据偏移量分别把程序段的数据读取到内存中

练习5：实现函数调用堆栈跟踪函数

我们需要在lab1中完成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。如果能够正确实现此函数，可在lab1中执行“make qemu”后，在qemu模拟器中得到类似如下的输出：

```

1 .....
2 ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58 0x001000

```

```

3      kern/debug/kdebug.c:305: print_stackframe+22
4      ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x00007b38
5      kern/debug/kmonitor.c:125: mon_backtrace+10
6      ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000 0x00007b38
7      kern/init/init.c:48: grade_backtrace2+33
8      ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4 0x00000000
9      kern/init/init.c:53: grade_backtrace1+38
10     ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000 0x00000000
11     kern/init/init.c:58: grade_backtrace0+23
12     ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x00000000
13     kern/init/init.c:63: grade_backtrace+34
14     ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007b38
15     kern/init/init.c:28: kern_init+88
16     ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502
17     <unknown>: -- 0x00007d72 -
18     .....

```

一般而言，ss:ebp+4处为返回地址，ss:ebp+8处为第一个参数值（最后一个入栈的参数值，此处假设其占用4字节内存），ss:ebp-4处为第一个局部变量，ss:ebp处为上一层ebp值。由于ebp中的地址处总是“上一层函数调用时的ebp值”，而在每一层函数调用中，都能通过当时的ebp值“向上（栈底方向）”能获取返回地址、参数值，“向下（栈顶方向）”能获取函数局部变量值。如此形成递归，直至到达栈底。

实验操作代码如下：

```

1  void
2  print_stackframe(void) {
3      /* LAB1 YOUR CODE : STEP 1 */
4      /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
5       * (2) call read_eip() to get the value of eip. the type is (uint32_t);
6       * (3) from 0 .. STACKFRAME_DEPTH
7       *   (3.1) printf value of ebp, eip
8       *   (3.2) (uint32_t)calling arguments [0..4] = the contents in address
9       *   (3.3) cprintf("\n");
10      *   (3.4) call print_debuginfo(eip-1) to print the C calling function
11      *   (3.5) popup a calling stackframe
12      *           NOTICE: the calling function's return addr eip = ss:[ebp+4]
13      *           the calling function's ebp = ss:[ebp]
14      */
15      uint32_t ebp = read_ebp(), eip = read_eip();
16      for (int i = 0; i < STACKFRAME_DEPTH && ebp != 0; i++) {
17          cprintf("ebp: 0x%08x eip: 0x%08x args:", ebp, eip);
18          for (int ij= 0; j < 4; j++) {
19              cprintf(" 0x%08x", ((uint32_t*)(ebp + 2))[j]);
20          }
21          cprintf("\n");
22          print_debuginfo(eip - 1);
23          eip = *((uint32_t*) ebp + 1);
24          ebp = *((uint32_t*) ebp);
25      }
26  }

```


练习6：完善中断初始化和处理

中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

中断描述符表中，一个表项占8字节，其中015位和4863为分别为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获得中断处理代码的入口。

请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt_init。

```

1 struct trapframe {
2     struct pushregs tf_regs;
3     uint16_t tf_gs;
4     uint16_t tf_padding0;
5     uint16_t tf_fs;
6     uint16_t tf_padding1;
7     uint16_t tf_es;
8     uint16_t tf_padding2;
9     uint16_t tf_ds;
10    uint16_t tf_padding3;
11    uint32_t tf_trapno;
12    /* 下面由x86硬件定义 */
13    uint32_t tf_err;
14    uintptr_t tf_eip;
15    uint16_t tf_cs;
16    uint16_t tf_padding4;
17    uint32_t tf_eflags;
18    /* 下面仅当交叉环时，例如从用户到内核 */
19    uintptr_t tf_esp;
20    uint16_t tf_ss;
21    uint16_t tf_padding5;
22 } __attribute__((packed));

```

trap.c中的print_ticks():

```

1 static void print_ticks() {
2     cprintf("%d ticks\n", TICK_NUM);
3     #ifdef DEBUG_GRADE volatile size_t ticks;
4     cprintf("End of Test.\n");
5     panic("EOT: kernel seems ok.");
6     #endif
7 }

```

clock.h中的ticks:

```

1 volatile size_t ticks;

```

trap.c中的TICK_NUM:

```

1 #define TICK_NUM 100

```

请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字”100 ticks”。

```
1 static void
2 trap_dispatch(struct trapframe *tf) {
3     char c;
4
5     switch (tf->tf_trapno) {
6     case IRQ_OFFSET + IRQ_TIMER:
7         ticks ++;
8         if (ticks % TICK_NUM == 0) {
9             print_ticks();
10        }
11        break;
```