

# 操作系统原理实验

## 实验二 - UCore启动过程

### 实验前准备

本次实验在Linux Ubuntu虚拟机环境下进行，主机环境为MacOS 10.14。实验报告使用Markdown语法，在Typora.app中编辑。

为满足实验要求，将命令行用户名PS1字段指定为姓名。PS1 = "许滨楠 -> "。

根据习惯将shell改为了zsh，可能在显示上会和默认有所出入。

### 练习1. 理解通过make生成执行文件的过程

#### 1. 操作系统镜像文件ucore.img是如何一步一步生成的？

根据实验要求，打开lab1代码目录，可以看见Makefile文件一共有洋洋洒洒的268行，逐行完成静态分析有一定难度，我们不妨先执行一次看看输出情况和结果。

首先，我们cd进入代码目录，为后续实验做准备：

```
1 | cd /home/moocos/moocos/ucore_lab/labcodes/lab1
```

```
moocos@moocos-VirtualBox: ~/moocos/ucore_lab/labcodes/lab1
许滨楠 -> cd /home/moocos/moocos/ucore_lab/labcodes/lab1
许滨楠 -> 
```

接着，先执行**make**，观察情况（较长的文字段采用复制展示，方便排版和查看）：

```
1 许滨楠 -> make
2 + cc kern/init/init.c
3 kern/init/init.c:95:1: warning: 'lab1_switch_test' defined but not used
  [-Wunused-function]
4   lab1_switch_test(void) {
5   ^
6 + cc kern/libs/readline.c
7 + cc kern/libs/stdio.c
8 + cc kern/debug/kdebug.c
9 kern/debug/kdebug.c:251:1: warning: 'read_eip' defined but not used [-
  Wunused-function]
10  read_eip(void) {
11  ^
12 + cc kern/debug/kmonitor.c
13 + cc kern/debug/panic.c
14 + cc kern/driver/clock.c
15 + cc kern/driver/console.c
16 + cc kern/driver/intr.c
17 + cc kern/driver/picirq.c
18 + cc kern/trap/trap.c
19 kern/trap/trap.c:14:13: warning: 'print_ticks' defined but not used [-
  Wunused-function]
20   static void print_ticks() {
21   ^
22 kern/trap/trap.c:30:26: warning: 'idt_pd' defined but not used [-Wunus
  ed-variable]
23   static struct pseudodesc idt_pd = {
24   ^
25 + cc kern/trap/trapentry.S
26 + cc kern/trap/vectors.S
27 + cc kern/mm/pmm.c
28 + cc libs/printfmt.c
29 + cc libs/string.c
30 + ld bin/kernel
31 + cc boot/bootasm.S
32 + cc boot/bootmain.c
```

```

33 + cc tools/sign.c
34 + ld bin/bootblock
35 'obj/bootblock.out' size: 472 bytes
36 build 512 bytes boot sector: 'bin/bootblock' success!
37 10000+0 records in
38 10000+0 records out
39 5120000 bytes (5.1 MB) copied, 0.0250776 s, 204 MB/s
40 1+0 records in
41 1+0 records out
42 512 bytes (512 B) copied, 0.000122463 s, 4.2 MB/s
43 138+1 records in
44 138+1 records out
45 70775 bytes (71 kB) copied, 0.000336849 s, 210 MB/s

```

这样，虽然还有不少看不懂的地方，但我们至少可以看到过程中相关的文件和其目录，以及大致的编译连接顺序，依赖关系。

接着，根据指引，我们可以利用make "V="指令，让make过程显示具体的执行语句，帮助我们更好地理解 and 定位语句位置，先通过**make clean**删除之前的结果，再执行**make "V="**，继续观察情况（此处的结果和语句对应行号将对我们接下来定位文件生成过程指令有重要作用）：

```

1 许滨楠 -> make clean
2  rm -f -r obj bin
3 许滨楠 -> make "V="
4  + cc kern/init/init.c
5  gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o
6  kern/init/init.c:95:1: warning: 'lab1_switch_test' defined but not used
   [-Wunused-function]
7      lab1_switch_test(void) {
8      ^
9  + cc kern/libs/readline.c
10 gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/libs/readline.c -o obj/kern/libs/readline.o
11 + cc kern/libs/stdio.c
12 gcc -Ikern/libs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/libs/stdio.c -o obj/kern/libs/stdio.o
13 + cc kern/debug/kdebug.c
14 gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o
15 kern/debug/kdebug.c:251:1: warning: 'read_eip' defined but not used [-
   Unused-function]
16     read_eip(void) {
17     ^
18 + cc kern/debug/kmonitor.c

```

```

19 gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o
20 + cc kern/debug/panic.c
21 gcc -Ikern/debug/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/debug/panic.c -o obj/kern/debug/panic.o
22 + cc kern/driver/clock.c
23 gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/driver/clock.c -o obj/kern/driver/clock.o
24 + cc kern/driver/console.c
25 gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/driver/console.c -o obj/kern/driver/console.o
26 + cc kern/driver/intr.c
27 gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/driver/intr.c -o obj/kern/driver/intr.o
28 + cc kern/driver/picirq.c
29 gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/driver/picirq.c -o obj/kern/driver/picirq.o
30 + cc kern/trap/trap.c
31 gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/trap/trap.c -o obj/kern/trap/trap.o
32 kern/trap/trap.c:14:13: warning: 'print_ticks' defined but not used [-
   Wunused-function]
33     static void print_ticks() {
34         ^
35 kern/trap/trap.c:30:26: warning: 'idt_pd' defined but not used [-Wunused-
   variable]
36     static struct pseudodesc idt_pd = {
37         ^
38 + cc kern/trap/trapentry.S
39 gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o
40 + cc kern/trap/vectors.S
41 gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/trap/vectors.S -o obj/kern/trap/vectors.o
42 + cc kern/mm/pmm.c
43 gcc -Ikern/mm/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-
   stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -
   Ikern/mm/ -c kern/mm/pmm.c -o obj/kern/mm/pmm.o
44 + cc libs/printfmt.c
45 gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-
   protector -Ilibs/ -c libs/printfmt.c -o obj/libs/printfmt.o
46 + cc libs/string.c

```

```

47 gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-
    protector -Ilibs/ -c libs/string.c -o obj/libs/string.o
48 + ld bin/kernel
49 ld -m elf_i386 -nostdlib -T tools/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
50 + cc boot/bootasm.S
51 gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-
    protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
52 + cc boot/bootmain.c
53 gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-
    protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
54 + cc tools/sign.c
55 gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
56 gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
57 + ld bin/bootblock
58 ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
    obj/boot/bootmain.o -o obj/bootblock.o
59 'obj/bootblock.out' size: 472 bytes
60 build 512 bytes boot sector: 'bin/bootblock' success!
61 dd if=/dev/zero of=bin/ucore.img count=10000
62 10000+0 records in
63 10000+0 records out
64 5120000 bytes (5.1 MB) copied, 0.0286946 s, 178 MB/s
65 dd if=bin/bootblock of=bin/ucore.img conv=notrunc
66 1+0 records in
67 1+0 records out
68 512 bytes (512 B) copied, 0.000122342 s, 4.2 MB/s
69 dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
70 138+1 records in
71 138+1 records out
72 70775 bytes (71 kB) copied, 0.000448696 s, 158 MB/s

```

有了以上基本的执行情况，就可以结合Makefile源文件进行探索。在探究ucore.img的生成过程时，我使用了"TOP-DOWN"的方法，自顶向下地从ucore.img的生成逐层向下查看依赖的文件和编译链接顺序。

首先，我们在Makefile中搜索ucore.img相关内容，很快能定位到Makefile中的178-188行：

```

1  # create ucore.img
2  UCOREIMG:= $(call totarget,ucore.img)
3
4  $(UCOREIMG): $(kernel) $(bootblock)
5  $(V)dd if=/dev/zero of=$@ count=10000
6  $(V)dd if=$(bootblock) of=$@ conv=notrunc
7  $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
8
9  $(call create_target,ucore.img)

```

不难看出，生成系统镜像文件的第一层依赖为**kernel**和**bootblock**两大块内容。

生成ucore.img时，实际运行的代码在make "V="结果中的61, 65, 69行：

```

1  dd if=/dev/zero of=bin/ucore.img count=10000
2  # 用于生成具有10000个单元块的文件
3  dd if=bin/bootblock of=bin/ucore.img conv=notrunc
4  # 写入bootblock相关内容
5  dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
6  # 写入kernel相关内容 (kernel内容seek定位到第1个单元块，bootblock写在了第0个单元块，
   这在bootmain.c的开头大段备注中的DISK LAYOUT也可以得到验证)

```

再向下一层，我们找到bootblock在Makefile中的相关重点内容，位于155-168行：

```

1  # create bootblock
2  bootfiles = $(call listf_cc,boot)
3  $(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -
   nostdinc))
4
5  bootblock = $(call totarget,bootblock)
6
7  $(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
8  @echo + ld $@
9  $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call
   toobj,bootblock)
10 @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
11 @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call
   outfile,bootblock)
12 @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
13
14 $(call create_target,bootblock)

```

根据其中的关键字，我们很容易地可以在make执行过程中定位生成bootblock的地方，位于make "V="结果中的57, 58行（对其中几个主要参数进行了解并注释）：

```

1 + ld bin/bootblock
2 ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
  obj/boot/bootmain.o -o obj/bootblock.o
3 # -m 对连接器的选择, 参数elf_i386表示选择模拟i386上的连接器
4 # -nostdlib 表示不使用标准库
5 # -N 设置了代码段、数据段均为可读写
6 # -e 指定entry, 入口为start
7 # -Ttext 指定代码段开始的位置为0x7C00

```

可以看出, 对于**bootblock**, 具有第二层依赖关系, 需要**bootmain.o**和**bootasm.o**两个文件, 经过**i386**模拟链接生成**bootblock.o**, 再由**objfile**生成**outfile**, 最后输出生成**bootblock**。在执行过程中, 还有**sign**文件的生成, 并且在Makefile 166行中也有**sign**文件的使用, 故判断**sign**可执行文件也是**bootblock**的依赖之一。

继续深入, 探究**bootmain.o**, **bootasm.o**和**sign**文件的生成, 查找执行过程, 定位到上面make结果的50-56行 (参数作用附于注释中) :

```

1 + cc boot/bootasm.S
2 gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-
  protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
3 + cc boot/bootmain.c
4 gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-
  protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
5 # -Iboot/ 指定搜索头文件的路径为boot/
6 # -fno-builtin 指定不进行builtin函数优化 (除非主动使用__builtin_前缀)
7 # -ggdb 生成可供gdb使用的调试信息
8 # -m32 生成适用32位环境的代码
9 # -gstabs 生成stabs格式的调试信息
10 # fno-stack-protector 指定不生成用于检测缓冲区溢出的代码
11 # -Os 指定优化, 减少代码的大小
12 # -nostdinc 不使用标准库
13 # -o 指定产生的结果目标文件
14 + cc tools/sign.c
15 gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
16 gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
17 # -g 编译的时候产生调试信息
18 # -Wall 启动所有Warning
19 # -O2 编译器优化级别为2 (-O0 表示没有优化, -O1为 缺省值, -O3 优化级别最高)
20 # -c 只产生编译的代码

```

不难判断, **bootasm.o**由**bootams.s**文件经过gcc编译产生; **bootmain.o**由**bootmain.c**文件经过gcc编译产生; **sign**由**sign.c**经过gcc编译产生**sign.o**进而产生。

**bootblock**这一块追根溯源探索完毕, 我们来看ucore.img的第二个依赖板块: **kernel**, 依然首先定位其Makefile相关代码段, 位于120-151行:

```

1 # kernel
2
3 KINCLUDE+= kern/debug/ \
4     kern/driver/ \

```

```

5     kern/trap/ \
6     kern/mm/
7
8     KSRCDIR+= kern/init \
9     kern/libs \
10    kern/debug \
11    kern/driver \
12    kern/trap \
13    kern/mm
14
15    KCFLAGS+= $(addprefix -I,$(KINCLUDE))
16
17    $(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))
18
19    KOBJS= $(call read_packet,kernel libs)
20
21    # create kernel target
22    kernel = $(call totarget,kernel)
23
24    $(kernel): tools/kernel.ld
25
26    $(kernel): $(KOBJS)
27    @echo + ld $@
28    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
29    @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
30    @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' >
    $(call symfile,kernel)
31
32    $(call create_target,kernel)

```

继续定位其实际执行命令，位于48-49行：

```

1  + ld bin/kernel
2  ld -m elf_i386 -nostdlib -T tools/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
3  # -T 指定连接器使用的脚本，此处定义为tools/kernel.ld
4  # 其他参数均于前述提及

```

可以看到kernel板块的第二层依赖主要为：init.o, kdebug.o, clock.o, picirq.o, vectors.o, string.o 几个文件，同时，根据文件目录结构，判断kern目录及libs目录下的文件生成均与kernel板块相关（对照参考答案亦是如此），按照老方法，我们顺藤摸瓜定位到上述文件的产生过程，因为实际指令过于冗长，且大同小异，不再贴上。位于make "V="结果的13-47行。且其中参数含义和作用均已在前面提及，不做赘述。Makefile倒是比较简短，定位到这几个文件的Makefile命令，位于Makefile中117-136行：



```
1 $(call add_files_cc,$(call listf_cc,$(LIBDIR)),libs,)
2 $(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))
```

各个.o文件均由其对应的.c文件经gcc编译产生。

至此，基本的文件依赖关系和ucore.img镜像文件的生成已经大具雏形，比较详细的生成过程已于此前说明，下面用较有层次结构的列表做一个总结罗列：

bin/ucore.img

bin/bootblock

obj/boot/bootasm.o

obj/boot/bootmain.o

bin/sign

bin/kernel

tools/kernel.ld

obj/kern/init/init.o

obj/kern/debug/kdebug.o

obj/kern/driver/clock.o

obj/kern/driver/picirq.o

obj/kern/trap/vectors.o

obj/libs/string.o

obj/libs/printfmt.o

obj/kern/libs/readline.o

obj/kern/libs/stdio.o

obj/kern/debug/kmonitor.o

obj/kern/debug/panic.o

obj/kern/driver/console.o

obj/kern/driver/intr.o

obj/kern/trap/trap.o

obj/kern/trap/trapentry.o

obj/kern/mm/pmm.o

## 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?

根据定义和/tools/sign.c代码中:

```
1 // 22行
2 char buf[512];
3 // 31-32行
4 buf[510] = 0x55;
5 buf[511] = 0xAA;
```

可知, 一个被系统认为是符合规范的硬盘主引导扇区的特征为: 1. 扇区具有512个字节; 2. 扇区最后两个字节分别为: 0x55和0xAA。

## 练习2. 使用qemu执行并调试lab1中的软件

### 1. 从CPU加电后执行的第一条指令开始, 单步跟踪BIOS的执行。

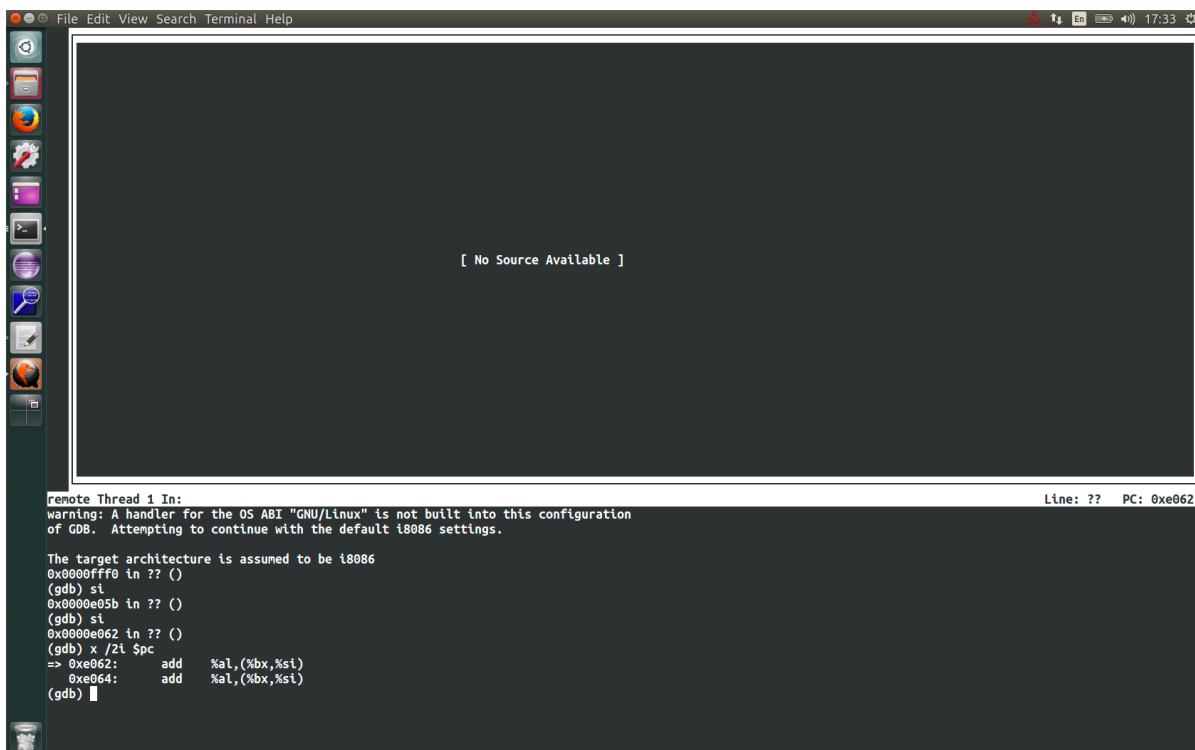
- 根据附录"启动后第一条执行的指令", 可知:
  - 第一条指令, 位于物理地址FFFFFFF0H, 将在硬件重启后被取出并执行。
  - 这个物理地址在处理器的最顶端物理地址的16个字节之后。
  - 该地址超过了实地址模式下的1MB的寻址空间。
  - $CS\ base\ address = CS\ segment\ selector * 16$
  - 实地址模式下的初始化过程, 是因为硬件重置将CS寄存器中的段选择子置为F000H, 而其将左移四位称为基地址, 所以基地址为FFF0000H, 加上偏移FFF0H, 第一条指令的物理地址是FFFFFFF0H。
  - 为保持CS寄存器的稳定, 引导程序/初始化软件程序应该不包括长距离的跳转或中断。
- 有了附录中的提示积累, 我们开始进行单步跟踪BIOS执行。
- 对lab1/tools/gdbinit进行修改, 修改后内容如下:

```
1 set architecture i8086
2 target remote : 1234
3 # 删除了continue, 防止qemu在gdb连接后马上开始执行。删除后qemu窗口将有[Stopped]标识
```

- cd进入lab1目录, 执行:

```
1 make debug
```

- 可以看到成功进入了gdb调试模式。此时从FFF0开始运行, 键入指令si/next/.....等可以进行跳转。
-



```
File Edit View Search Terminal Help
[ No Source Available ]

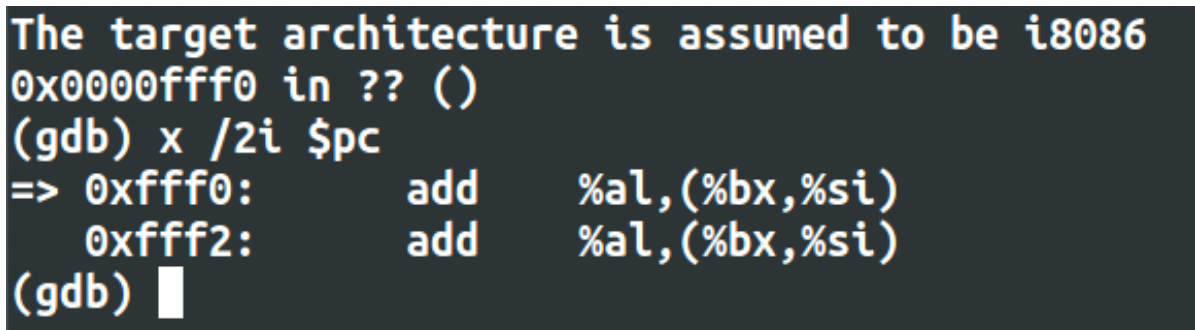
remote Thread 1 In:
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
0x0000ffff in ?? ()
(gdb) si
0x0000e05b in ?? ()
(gdb) si
0x0000e062 in ?? ()
(gdb) x /2i $pc
=> 0xe062:      add    %al, (%bx,%si)
    0xe064:      add    %al, (%bx,%si)
(gdb) █
```

- 但此时我们还无法看到具体的指令，只能看到PC中的代码段指针位置。键入以下指令可以看到当前运行中的汇编指令。

```
1 | x /2i $pc
```

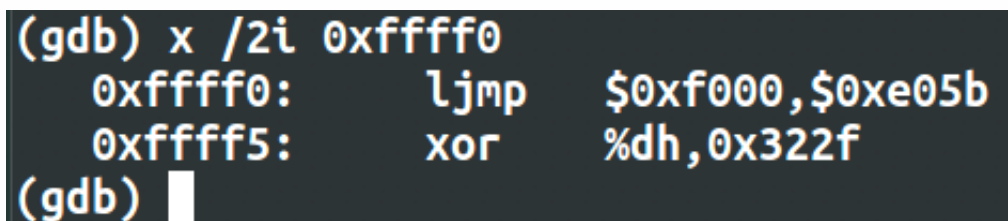
- 查看第一条指令时，发现其指令反汇编为：



```
The target architecture is assumed to be i8086
0x0000ffff in ?? ()
(gdb) x /2i $pc
=> 0xffff0:      add    %al, (%bx,%si)
    0xffff2:      add    %al, (%bx,%si)
(gdb) █
```

- 分析原因是，此时的\$pc并非程序计数器，而是%eip中的值，所以会出现第一条指令不是跳转指令的情况。根据附件中我们已经解读到的知识，0xFFFF0才应该是第一条指令，键入以下指令查看该处信息：

```
1 | x /i 0xffff0
```



```
(gdb) x /2i 0xffff0
0xffff0:      ljmp    $0xf000, $0xe05b
0xffff5:      xor     %dh, 0x322f
(gdb) █
```

是跳转指令无误，其地址为前文提及的 $(CS \ll 4) | (EIP) = 0xFFFF0$ 。

- 我们还可以通过在刚才的gdbinit文件中增加以下代码，实现将每条指令反汇编成汇编指令，每次显示。

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

- 可以看到，现在我们可以实时观察到汇编命令的执行了。

■



## 2. 在初始化位置0x7c00设置实地址断点，测试断点正常。

- gdb设置断点的指令为 `b *address`，因此为满足题意，我们需要在gdb启动后键入 `b *0x7c00`，接着键入 `c` 让程序继续执行，就可以一直运行到断点位置。再在断点处键入指令查看前十条指令，结果如下：

```

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x /10i $pc
=> 0x7c00:      cli
    0x7c01:      cld
    0x7c02:      xor      %ax,%ax
    0x7c04:      mov      %ax,%ds
    0x7c06:      mov      %ax,%es
    0x7c08:      mov      %ax,%ss
    0x7c0a:      in       $0x64,%al
    0x7c0c:      test     $0x2,%al
    0x7c0e:      jne      0x7c0a
    0x7c10:      mov      $0xd1,%al
(gdb) █

```

- 正常。且经测试，将以上指令直接在运行debug之前写入gdbinit文件，可以达到同样的效果。

3. 在调用qemu时增加-d in\_asm -D q.log参数，便可以将运行的汇编指令保存在q.log中。将执行的汇编代码与bootasm.S和bootblock.asm进行比较，看看二者是否一致。

- 在Makefile的debug部分做出一些修改，添加-d in\_asm -D q.log参数，将汇编指令保存在/bin/q.log文件里面。

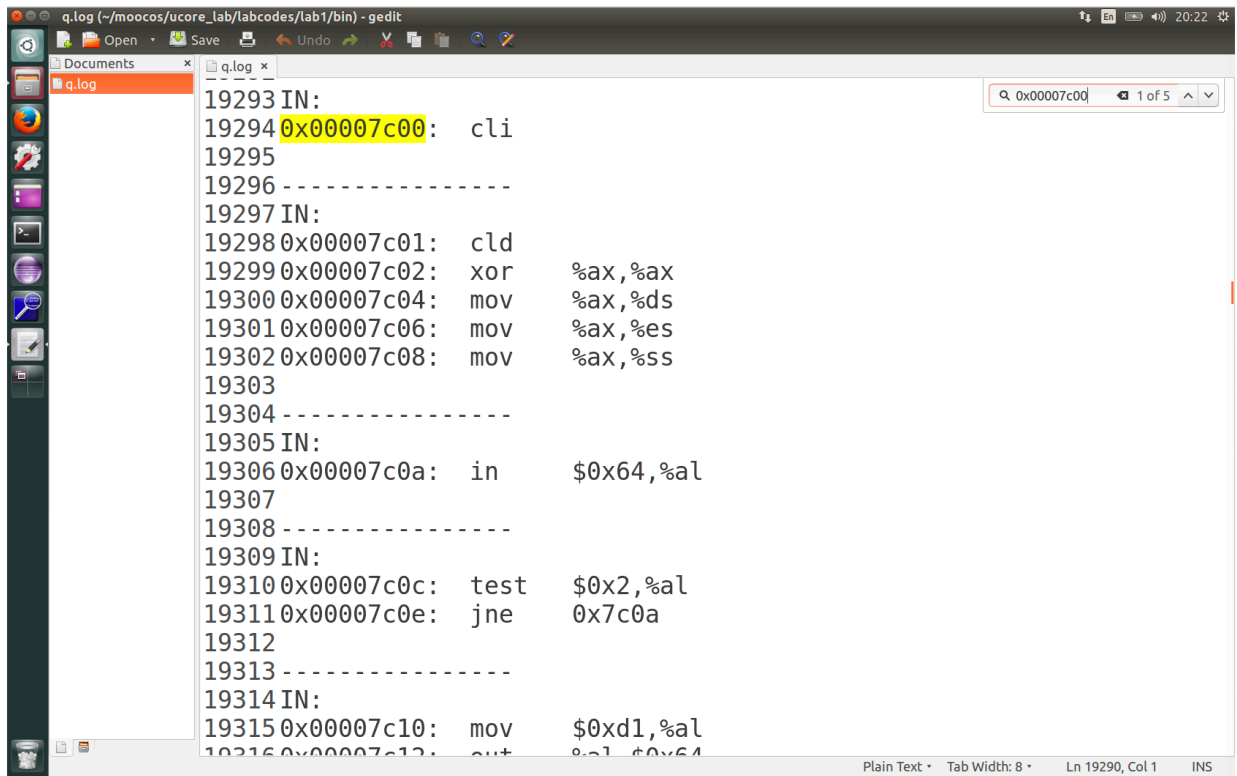
```

1  debug: $(UCOREIMG)
2  $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -parallel
   stdout -hda $< -serial null"
3  $(V)sleep 2
4  $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"

```

- 运行gdb之后，键入b \*0x7c00设置断点，再键入c继续运行程序至断点处。用指令x /10i \$pc查看指令，断点工作正常，此时再进入c继续执行。
- 然后我们就可以在bin目录中的q.log中查看执行过的命令，搜索到0x00007c00处，指令运行情况如下：

-



```
q.log (~/.moocos/ucore_lab/labcodes/lab1/bin) - gedit
Documents
q.log
19293 IN:
19294 0x00007c00: cli
19295
19296 -----
19297 IN:
19298 0x00007c01: cld
19299 0x00007c02: xor    %ax,%ax
19300 0x00007c04: mov    %ax,%ds
19301 0x00007c06: mov    %ax,%es
19302 0x00007c08: mov    %ax,%ss
19303
19304 -----
19305 IN:
19306 0x00007c0a: in     $0x64,%al
19307
19308 -----
19309 IN:
19310 0x00007c0c: test   $0x2,%al
19311 0x00007c0e: jne    0x7c0a
19312
19313 -----
19314 IN:
19315 0x00007c10: mov    $0xd1,%al
19316 0x00007c12: out    %al,$0x64

Plain Text • Tab Width: 8 • Ln 19290, Col 1 • INS
```

- 对照lab1/boot/目录下bootasm.S和 bootblock.asm进行比对，除了因反汇编不会给指令加入l, w等指示操作数大小的后缀意外，q.log中的指令情况和bootasm.S与bootblock.asm相同。

### 练习3. 分析bootloader进入保护模式的过程

通过实验指导中的知道意见，阅读3.2.1 保护模式和分段机制，结合lab1/boot/bootasm.S源代码及其中注释，做出以下分析，分析如何从实模式切换到保护模式。

我们从头开始逐一解读。

在代码头文件下面紧接着的注释中我们得知：

CPU启动时，将切换到32位保护模式，跳转到C。BIOS将从硬盘的第一块载入这段代码到内存中的0x7c00并开始在实模式下执行。当前%cs=0，%ip=7c00。

接着是一段宏定义：

```
1 .set PROT_MODE_CSEG,      0x8           # kernel code segment
  selector
2 .set PROT_MODE_DSEG,      0x10          # kernel data segment
  selector
3 .set CR0_PE_ON,           0x1           # protected mode
  enable flag
```

易知分别定义了内核代码段选择子、数据段选择子以及保护模式标志位。

再接下来是寄存器的初始化工作。"清理"运行环境。并开启A20。

A20是第21条地址线上的一个开关，根据附录中"关于A20 Gate"的参考资料，这个开关的出现是为了在16M内存的80286和4G内存的80386出现时实现与8086的兼容，用A20 Gate来模拟8086机上寻址越界时的"回卷现象"。这个开关控制了这条地址线（也即A20）的可用与否，当开关打开，这条地址线正常工作；当开关关闭，A20恒为0，这将导致系统只能访问奇数M的内存，"这显然是不行的，所以在保护模式下，这个开关也必须打开"。同时，当A20 Gate为关闭状态，也将导致1MB以上的地址是不可寻址的，这就让内存更大，寻址能力本应更强的80286和80386失去了优势，"为了使能所有地址位的寻址能力，必须向键盘控制器 8042 发送一个命令。键盘控制器 8042 将会将它的的某个输出引脚的输出置高电平，作为 A20 地址线控制的输入。"（附录中还说明了其实A20 Gate和8042如我们乍看之下的感觉一样，没有一点关系，推测是早期设计师为了节约硬件成本设计。）因此有了下面代码：

```
1 seta20.1:
2     inb $0x64, %al    # Wait for not busy(8042 input buffer empty).
3     testb $0x2, %al
4     jnz seta20.1
5
6     movb $0xd1, %al    # 0xd1 -> port 0x64
7     outb %al, $0x64    # 0xd1 means: write data to 8042's P2 port
8
9 seta20.2:
10    inb $0x64, %al    # Wait for not busy(8042 input buffer empty).
11    testb $0x2, %al
12    jnz seta20.2
13
14    movb $0xdf, %al    # 0xdf -> port 0x60
15    outb %al, $0x60    # 0xdf = 11011111, means set P2's A20 bit(the 1
                        bit) to 1
```

具体内容注释中已经非常详尽，基本步骤为：

1. 等待8042的input buffer为idle状态；
2. 向8042的64h端口发送"向P2端口写数据"的命令；
3. 等待8042的input buffer为idle状态；
4. 向8042的60h端口发送"设置P2端口的A20位置为1"的命令。

完成这一步之后，A20 Gate已经打开，32条地址线全部可以正常工作，可以访问到4G的内存空间。

接着是GDT（全局描述符表）的初始化以及保护模式的开启代码：

```
1 # Switch from real to protected mode, using a bootstrap GDT
2 # and segment translation that makes virtual addresses
3 # identical to physical addresses, so that the
4 # effective memory map does not change during the switch.
5 lgdt gtdesc
6 movl %cr0, %eax
7 orl $CR0_PE_ON, %eax
8 movl %eax, %cr0
9
10 # Bootstrap GDT
```

```

11  .p2align 2                                # force 4 byte
    alignment
12  gdt:
13      SEG_NULLASM                            # null seg
14      SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg for
    bootloader and kernel
15      SEG_ASM(STA_W, 0x0, 0xffffffff)        # data seg for
    bootloader and kernel
16
17  gdtdesc:
18      .word 0x17                             # sizeof(gdt) - 1
19      .long gdt                             # address gdt

```

初始的GDT表已经存储在引导区中，直接通过描述符载入即可。

接着将宏定义中的保护模式标志CR寄存器（0x1）的低位置为1，进入保护模式。

```

1  # Jump to next instruction, but in 32-bit code segment.
2  # Switches processor into 32-bit mode.
3  ljmp $PROT_MODE_CSEG, $protcseg

```

练习2的第一题中，我们说过附录"启动后第一条执行的指令"说明：长跳转或中断调用将导致CS寄存器的改变。此处进行的长跳转将导致CS寄存器的值被修改为宏定义中的保护模式PROT\_MODE\_CSEG(0x8)。

接下来的代码主要是设置保护模式的数据段寄存器并建立堆栈区，主要包含了DS、ES、FS、GS、SS等段寄存器的初始化设定，完成后跳入bootmain段。（最后还有一个死循环的设定，防止程序错误执行至此（并不会），非常具有操统风。）

```

1  .code32                                    # Assemble for 32-bit
    mode
2  protcseg:
3      # Set up the protected-mode data segment registers
4      movw $PROT_MODE_DSEG, %ax            # Our data segment
    selector
5      movw %ax, %ds                        # -> DS: Data Segment
6      movw %ax, %es                        # -> ES: Extra
    Segment
7      movw %ax, %fs                        # -> FS
8      movw %ax, %gs                        # -> GS
9      movw %ax, %ss                        # -> SS: Stack
    Segment
10
11     # Set up the stack pointer and call into C. The stack region is from
    0--start(0x7c00)
12     movl $0x0, %ebp
13     movl $start, %esp
14     call bootmain
15
16     # If bootmain returns (it shouldn't), loop.
17 spin:

```



## 练习4. 分析bootloader加载ELF格式的OS的过程

bootmain.c中有详尽的备注和显式的函数名提示，可读性还是比较高的，下面针对两个具体问题，结合其中实际的c语言代码进行分析。

### 1. bootloader是如何读取硬盘扇区的？

读取硬盘扇区对应的代码如下：

```
1  /* waitdisk - wait for disk ready */
2  static void
3  waitdisk(void) {
4      while ((inb(0x1F7) & 0xC0) != 0x40)
5          /* do nothing */;
6  }
7
8  /* readsect - read a single sector at @secno into @dst */
9  static void
10 readsect(void *dst, uint32_t secno) {
11     // wait for disk to be ready
12     waitdisk();
13
14     outb(0x1F2, 1);                // count = 1
15     outb(0x1F3, secno & 0xFF);
16     outb(0x1F4, (secno >> 8) & 0xFF);
17     outb(0x1F5, (secno >> 16) & 0xFF);
18     outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
19     outb(0x1F7, 0x20);            // cmd 0x20 - read sectors
20
21     // wait for disk to be ready
22     waitdisk();
23
24     // read a sector
25     insl(0x1F0, dst, SECTSIZE / 4);
26 }
27
28 /* *
29  * readseg - read @count bytes at @offset from kernel into virtual
30  * address @va,
31  * might copy more than asked.
32  * */
32 static void
33 readseg(uintptr_t va, uint32_t count, uint32_t offset) {
```

```

34     uintptr_t end_va = va + count;
35
36     // round down to sector boundary
37     va -= offset % SECTSIZE;
38
39     // translate from bytes to sectors; kernel starts at sector 1
40     uint32_t secno = (offset / SECTSIZE) + 1;
41
42     // If this is too slow, we could read lots of sectors at a time.
43     // We'd write more to memory than asked, but it doesn't matter --
44     // we load in increasing order.
45     for (; va < end_va; va += SECTSIZE, secno++) {
46         readsect((void *)va, secno);
47     }
48 }

```

其中waitdisk函数将读取磁盘的IO状态和命令寄存器，判断磁盘当前是否可用。

接下来的readsect函数中，在判断可以开始读磁盘时，先设置读写的扇区数。通过连续的四条outb指令，设置LBA模式下的参数。具体的IO地址和相应的功能如下：

表一 磁盘 IO 地址和对应功能

IO 地址	功能
0x1f0	读数据，当 0x1f7 不为忙状态时，可以读。
0x1f2	要读写的扇区数，每次读写前，你需要表明你要读写几个扇区。最小是 1 个扇区
0x1f3	如果是 LBA 模式，就是 LBA 参数的 0-7 位
0x1f4	如果是 LBA 模式，就是 LBA 参数的 8-15 位
0x1f5	如果是 LBA 模式，就是 LBA 参数的 16-23 位
0x1f6	第 0~3 位：如果是 LBA 模式就是 24-27 位      第 4 位：为 0 主盘；为 1 从盘 第 6 位：为 1=LBA 模式；0=CHS 模式      第 7 位和第 5 位必须为 1
0x1f7	状态和命令寄存器。操作时先给命令，再读取，如果不是忙状态就从 0x1f0 端口读数据

可知其设置情况为：29-31位为1，28位为0标识访问Disk 0，0-27位作为28位的偏移量。

接着，向状态和命令寄存器0x1f7传送0x20，读取扇区的命令。读取目的地dst。

readseg函数是对readsect的包装，通过控制参数offset偏移量，count读取字节数量，va目的地址的传递进行读取。每次读取以扇区为单位，并且因为0号扇区被引导程序占用，从一号扇区开始。

## 2. bootloader如何加载ELF格式的OS?

对于ELF格式的OS的载入，主要代码位于bootmain函数中：

```

1  /* bootmain - the entry of bootloader */
2  void
3  bootmain(void) {
4      // read the 1st page off disk
5      readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
6
7      // is this a valid ELF?

```

```

8     if (ELFHDR->e_magic != ELF_MAGIC) {
9         goto bad;
10    }
11
12    struct proghdr *ph, *eph;
13
14    // load each program segment (ignores ph flags)
15    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
16    eph = ph + ELFHDR->e_phnum;
17    for (; ph < eph; ph++) {
18        readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
19    }
20
21    // call the entry point from the ELF header
22    // note: does not return
23    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();
24
25    bad:
26        outw(0x8A00, 0x8A00);
27        outw(0x8A00, 0x8E00);
28
29        /* do nothing */
30        while (1);
31    }

```

这个函数从磁盘读取ELF的第一页，通过用以标记文件或者协议的格式的magic数判断该ELF是否合法，若不合法直接会跳转到置位和经典的while(1)死循环处理，合法则进行解析，先得到首地址和大小，然后将ELF文件中数据载入内存，最后跳转到内核入口执行。

## 练习5. 实现函数调用堆栈跟踪函数

完成kdebug.c中函数print\_stackframe的实现

```

1    void
2    print_stackframe(void) {
3        uint32_t ebp = read_ebp();
4        uint32_t eip = read_eip();
5        uint32_t args[4];
6        for (int i = 0; i < STACKFRAME_DEPTH && ebp; ++i) {
7            for (int j = 0; j < 4; ++j)
8                args[j] = *(uint32_t*)(ebp + 8 + 4 * j);
9            cprintf("ebp:0x%08x eip:0x%08x args:0x%08x 0x%08x 0x%08x\n",
10                ebp, eip, args[1], args[2], args[3]);
11            print_debuginfo(eip - 1);
12            eip = *(uint32_t*)(ebp + 4);

```

```

13     ebp = *(uint32_t*)ebp;
14 }
15 }

```

根据代码中备注的提示，写出代码如上。运行make qemu进行验证，得输出结果如下：

```

Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00000000 0x00007b38 0x00100092 0x00007b18
    kern/debug/kdebug.c:306: print_stackframe+21
ebp:0x00007b18 eip:0x00100c8d args:0x00000000 0x00000000 0x00007b88 0x00007b18
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00007b60 0xffff0000 0x00007b64 0x00007b18
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0xffff0000 0x00007b84 0x00000029 0x00007b18
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00100000 0xffff0000 0x0000001d 0x00007b18
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x001032c0 0x0000130a 0x00000000 0x00007b18
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00010094 0x00007b18
    kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc08ed88e 0x64e4d08e 0xfa7502a8 0x00007b18
    <unknown>: -- 0x00007d67 --
++ setup timer interrupts
许滨楠 -> █

```

基本符合要求。（源代码讲一同打包提交，下同。）

最后一行对应的是堆栈中最“深”的一层，是bootmain.c中的bootmain函数。ebp:0x00007bf8是该函数的ebp，bootasm中已说明开始时esp=0x7c00，ebp=0，调用bootmain函数时返回值和ebp压栈，0x7c00-0x8=0x00007bf8。最后的参数为大于0x7c00地址中的内容。

## 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多少字节？其中哪几位代表中断处理代码的入口？

中断向量表IDT中一个表项占8个字节，其中第2,3个字节是段选择子，0,1字节和6,7字节可以拼成offset。通过段选择子查找段描述符表，结合offset便可以确定中断处理代码的入口地址。

2. 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。

根据题目提示，在函数中需要对所有中断入口进行初始化。根据注释引导和指示，完成代码如下：

```

1 void
2 idt_init(void) {
3     extern uintptr_t __vectors[];
4     int i;
5     for (i = 0; i < 256; ++i) {
6         SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
7     }
8     SETGATE(idt[T_SWITCH_TOK], 1, GD_KTEXT, __vectors[T_SWITCH_TOK],
DPL_USER);
9     lidt(&idt_pd);
10 }

```

先定位中断入口，然后对IDT进行填充，完毕之后进行加载。

### 3. 编程完善trap.c中的中断处理函数trap。

根据注释及答案的提示（PS: to be honest严重参考了参考答案...），完成代码如下：

```

1  /* temporary trapframe or pointer to trapframe */
2  struct trapframe switchk2u, *switchu2k;
3
4  /* trap_dispatch - dispatch based on what type of trap occurred */
5  static void
6  trap_dispatch(struct trapframe *tf) {
7      char c;
8
9      switch (tf->tf_trapno) {
10         case IRQ_OFFSET + IRQ_TIMER:
11             /* LAB1 YOUR CODE : STEP 3 */
12             /* handle the timer interrupt */
13             /* (1) After a timer interrupt, you should record this event
using a global variable (increase it), such as ticks in
kern/driver/clock.c
14             * (2) Every TICK_NUM cycle, you can print some info using a
funciton, such as print_ticks().
15             * (3) Too Simple? Yes, I dooooooon't think so!
16             */
17             ticks ++;
18             if (ticks % TICK_NUM == 0) {
19                 print_ticks();
20             }
21             break;
22         case IRQ_OFFSET + IRQ_COM1:
23             c = cons_getc();
24             cprintf("serial [%03d] %c\n", c, c);
25             break;
26         case IRQ_OFFSET + IRQ_KBD:
27             c = cons_getc();
28             cprintf("kbd [%03d] %c\n", c, c);

```

```

29         break;
30         //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
31         case T_SWITCH_T0U:
32             if (tf->tf_cs != USER_CS) {
33                 switchk2u = *tf;
34                 switchk2u.tf_cs = USER_CS;
35                 switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss =
USER_DS;
36                 switchk2u.tf_esp = (uint32_t)tf + sizeof(struct
trapframe) - 8;
37
38                 // set eflags, make sure ucore can use io under user
mode.
39                 // if CPL > IOPL, then cpu will generate a general
protection.
40                 switchk2u.tf_eflags |= FL_IOPL_MASK;
41
42                 // set temporary stack
43                 // then iret will jump to the right stack
44                 *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
45             }
46             break;
47         case T_SWITCH_T0K:
48             if (tf->tf_cs != KERNEL_CS) {
49                 tf->tf_cs = KERNEL_CS;
50                 tf->tf_ds = tf->tf_es = KERNEL_DS;
51                 tf->tf_eflags &= ~FL_IOPL_MASK;
52                 switchu2k = (struct trapframe *) (tf->tf_esp -
(sizeof(struct trapframe) - 8));
53                 memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
54                 *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
55             }
56             break;
57         case IRQ_OFFSET + IRQ_IDE1: break;
58         case IRQ_OFFSET + IRQ_IDE2: break;
59         default:
60             // in kernel, it must be a mistake
61             if ((tf->tf_cs & 3) == 0) {
62                 print_trapframe(tf);
63                 panic("unexpected trap in kernel.\n");
64             }
65     }
66 }
67
68 /* *
69  * trap - handles or dispatches an exception/interrupt. if and when
trap() returns,
70  * the code in kern/trap/trapentry.S restores the old CPU state saved in
the
71  * trapframe and then uses the iret instruction to return from the
exception.
72  * */

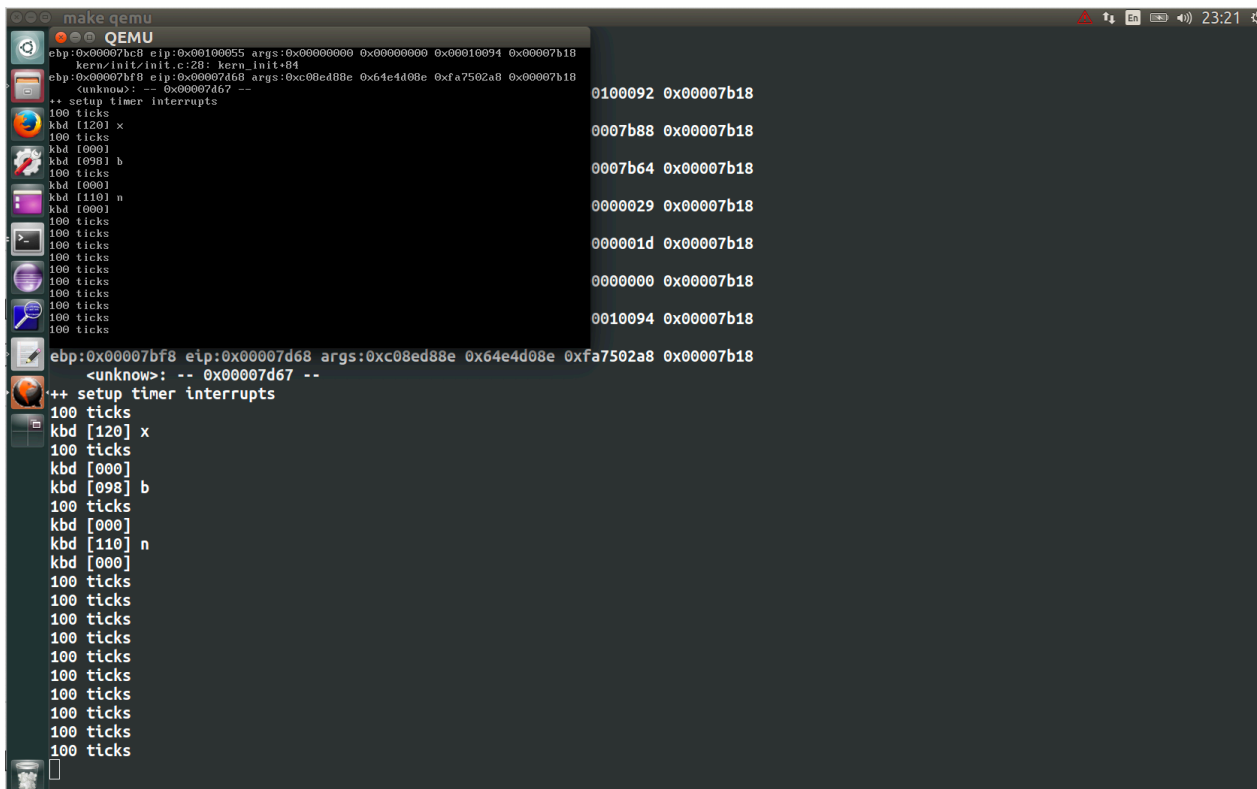
```

```

73 void
74 trap(struct trapframe *tf) {
75     // dispatch based on what type of trap occurred
76     trap_dispatch(tf);
77 }

```

对2, 3题进行验证，运行系统。



在目录下键入make qemu命令，运行系统，可以看到屏幕上以大约一秒一次的频率输出100 ticks，同时系统会响应键盘输入。基本符合要求。

## 实验心得体会

本次实验，真的很难！

至少从第一次的AT&T汇编到现在，难度的加大可以说是非常明显了。但这也是一开学老师就给我们打过预防针的了吧。老师说操作系统就是枯燥的理论课加上很难的实验课。但经过一个多月的学习，感觉上其实操作系统的理论课还算是比较有趣的，老师的讲解也比较清楚。但是实验确实很难。

本次实验的内容比较多，光是参考资料前前后后就看了接近一天，并且因为内容还是比较多，有些内容有一定理解难度，有些内容又比较枯燥，所以看的时候进度非常慢，而且容易遗忘，很多点到了做实验的时候只留下一点可供查找的小小印象，还是要频频翻阅参考资料，上网搜索相关信息。这应该是学习状态和方法上需要改进的问题。

得益于充足的参考资料和比较有参考价值的答案解析，这次的实验内容从一开始完全地看不懂到最后慢慢地还是啃下来了。虽然还有有心无力的challenge没有好好做完，但回过头来觉得上面的几个练习都走下来了还是比较意外的。其中有一开始就吓到我，但是后来发现不需要刨根问底而是需要准确筛选定位信息就能解决的Makefile分析；也有很久以前就接触过但一直没有机会运用的GDB调试；还有结合

代码注释和提醒对代码的阅读理解等等。这些问题虽然只是给了我们一个管中窥豹的机会，但是也对于操作系统这个大家伙，其实也算是可见一斑了。平常学到的寻址、中断、引导程序等内容，都在这次的实验中从理论走向了可见的代码和程序软件，对于操作系统原理实验的学习，这样的一次探索应该算是一个良好的开端。

本来感慨这么多看不懂的地方，着实需要老师在实验中的更多引导，但是回过头来发现，其实都是一些分析代码，信息检索和整合的基本能力，锻炼的机会比得到指导的机会更加难得。所以感觉更应好好锻炼相关的能力，这对以后的学习和技能应用都是至关重要的。

另外还有一点感悟比较深的是，因为对这次的实验没有很大的把握，所以不敢自己配置虚拟机，只好直接用VirtualBox导入vdi文件。但这样生成的虚拟机实在是太过卡顿了。因为调整分辨率之后会有改善，所以判断应该是虚拟机显卡驱动的问题，但是暂时没有找到解决方案。所以要稍微正常地使用虚拟机只能把分辨率调到只有自己电脑六分之一屏幕大小的一个小窗中使用，确实不太方便。实验做完了有时间再好好看看如何解决这个问题。

囫圇吞枣式的入门探索告一段落，总结起来还是觉得应该在过后再花时间好好消化，不能只停留在答完了这几个问题，因为有参考答案的引导所以其实很多地方没有真正掌握而只是随波逐流地去做。同志仍需努力。