

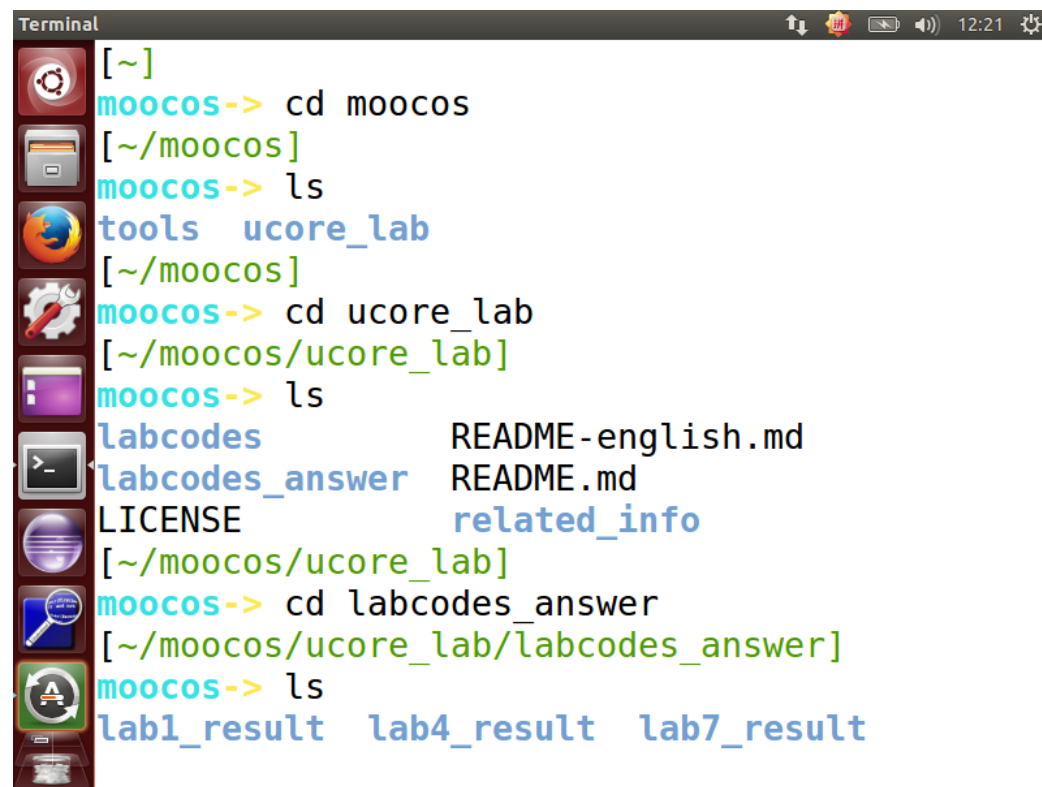
# lab1

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命令行中，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**

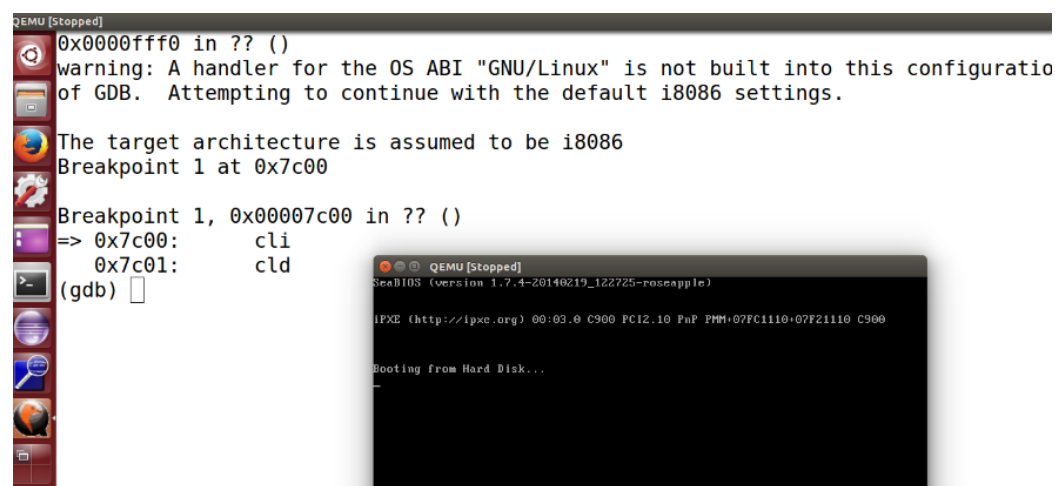
设置断点进行测试：

```
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。