

# lab1 实验报告

## 练习一

### 题目

理解通过make生成执行文件的过程。（要求在报告中写出对下述问题的回答）

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：

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）

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？  
123

### 解答

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）

### 通过make V= 生成的make 生成过程

附带注释，只是目前为止本人的理解，不保证正确性。

```
1
cd lab1 # 进入lab1文件夹中
make clean # 清除之前生成的项目
make V= # 获取make 的生成过程
123
```

### 生成过程如下

```
+ cc kern/init/init.c # 编译ucore系统内核启动文件 init.c
gcc -Ikern/init/ -march=i686 -fno-builtin -fno-PIC -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
kern/init/init.c:95:1: warning: 'lab1_switch_test' defined but not used [-Wunused-function]
   95 | lab1_switch_test(void) {
      | ^~~~~~
+ cc kern/libs/stdio.c # 编译标准输入输出库
gcc -Ikern/libs/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/libs/readline.c # 一个用于(从stdin中)读取一行内容的方法
```

```

gcc -Ikern/libs/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/debug/panic.c # 一个用于发出报错的方法（不确定，现阶段的理解）
gcc -Ikern/debug/ -march=i686 -fno-builtin -fno-PIC -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
kern/debug/panic.c: In function '__panic':
kern/debug/panic.c:27:5: warning: implicit declaration of function
'print_stackframe'; did you mean 'print_trapframe'? [-Wimplicit-function-
declaration]
    27 |     print_stackframe();
        |     ^~~~~~
        |     print_trapframe
+ cc kern/debug/kdebug.c # 包含几个debug 方法，打印信息，二分查找地址等
gcc -Ikern/debug/ -march=i686 -fno-builtin -fno-PIC -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
kern/debug/kdebug.c:251:1: warning: 'read_eip' defined but not used [-Wunused-
function]
    251 | read_eip(void) {
        | ^~~~~~
+ cc kern/debug/kmonitor.c # 简单的命令行内核监视器
gcc -Ikern/debug/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/driver/clock.c # 好像是时钟中断信号
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/driver/console.c # 初始化显示器，连续内存（不确定）
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/driver/picirq.c # 初始化程序中断控制器 pic（不确定）
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/driver/intr.c # irq 的 enable 和disable
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/trap/trap.c # 暂未深入研究
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -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
kern/trap/trap.c: In function 'print_trapframe':
kern/trap/trap.c:100:16: warning: taking address of packed member of 'struct
trapframe' may result in an unaligned pointer value [-Waddress-of-packed-member]
    100 |     print_regs(&tf->tf_regs);
        |             ^~~~~~
At top level:
kern/trap/trap.c:30:26: warning: 'idt_pd' defined but not used [-Wunused-
variable]
    30 | static struct pseudodesc idt_pd = {

```

```

|                                     ^~~~~~
kern/trap/trap.c:14:13: warning: 'print_ticks' defined but not used [-Wunused-
function]
    14 | static void print_ticks() {
        |             ^~~~~~
+ cc kern/trap/vectors.S # 一段汇编代码，还未研究是干吗的
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/trap/trapentry.S # 一段汇编代码，还未研究是干吗的
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -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
+ cc kern/mm/pmm.c # 好像跟中断描述符有关
gcc -Ikern/mm/ -march=i686 -fno-builtin -fno-PIC -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
+ cc libs/string.c # String 字符串依赖
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
-fno-stack-protector -Ilibs/ -c libs/string.c -o obj/libs/string.o
+ cc libs/printfmt.c # 好像是格式化输出依赖
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
-fno-stack-protector -Ilibs/ -c libs/printfmt.c -o obj/libs/printfmt.o
+ ld bin/kernel #将上面编译的文件转化为可执行文件，即系统内核存储在 bin/kernel中
ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel
obj/kern/init/init.o obj/kern/libs/stdio.o obj/kern/libs/readline.o
obj/kern/debug/panic.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o
obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/picirq.o
obj/kern/driver/intr.o obj/kern/trap/trap.o obj/kern/trap/vectors.o
obj/kern/trap/trapentry.o obj/kern/mm/pmm.o obj/libs/string.o
obj/libs/printfmt.o
+ cc boot/bootasm.S # 编译bootasm.S文件，这个是当前lab中 Bootloader的一部分，其主要目的
是被cpu调用，然后该文件调用bootmain.c文件， 同时该文件还将系统设置成了保护模式，以及一个栈用于
c语言代码的调用
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
-fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o
obj/boot/bootasm.o
+ cc boot/bootmain.c # 编译bootmain.c文件
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
-fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o
obj/boot/bootmain.o
+ cc tools/sign.c # 编译sign.c文件 该文件好像是用于获取符合规范的磁盘主引导扇区的内容，及
将磁盘主引导扇区的内容加载到内存中，同时还有校验主引导扇区中的内容是否符合要求（不确定）
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
+ ld bin/bootblock # 将上面生成的*.o文件通过ld命令链接成bin 目录下的bootblock文件
ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 496 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000 # /dev/zero 作为输入文件的作用是将无数
个0写入文件（这边是10000个）
记录了10000+0 的读入
记录了10000+0 的写出
5120000字节（5.1 MB, 4.9 MiB）已复制，0.0313203 s, 163 MB/s

```

```
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512字节已复制，0.000126482 s，4.0 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
记录了154+1 的读入
记录了154+1 的写出
78916字节（79 kB，77 KiB）已复制，0.000435106 s，181 MB/s # 以上两个dd 命令将之前生成的
bootblock 和 kernel写入到ucore.img中 这时这个虚拟磁盘镜像就生成了
12345678910111213141516171819202122232425262728293031323334353637383940414243444
546474849505152535455565758596061626364656667686970717273747576777879
```

## /dev/zero文件的描述

/dev/zero 是类 Unix 系统中一个特殊的文件，当读取该文件时，它会提供无限的空字符 `null`。它的一个主要用途是提供字符流来初始化数据存储，也就是使用空字符覆盖目标数据。另一个常见的用法是产生一个特定大小的空白文件。

你可以从 /dev/zero 读取任意大小数量的 `null` 字符。和 /dev/null 不同，/dev/zero 不但可以作为数据黑洞，也可以作为数据源泉。你可以将数据写入 /dev/zero 文件，但实际上不会有任何影响。不过一般我们还是使用 /dev/null 来做这件事。

123

## 总结ucore.img操作系统镜像生成的过程

下面内容有借鉴其他人的答案（如：

[https://blog.csdn.net/qg\\_41946412/article/details/103091369](https://blog.csdn.net/qg_41946412/article/details/103091369)）

1. 通过gcc编译器，将kernel文件夹下的.c文件编译成Obj目录下的.o文件
2. 通过ld命令以及提供给我们的kernel.ld脚本文件将.o文件 链接为bin目录下的kernel可执行文件，且该文件是一个ELF格式的 32位可执行文件 具体如下所示

```
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes/lab1/bin$ file
kernel
```

```
kernel: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically
linked, not stripped
```

#该文件可以说是ucore系统内核文件（目前我的理解）

12345

1. 然后将boot目录下的.c.S文件 以及tools下的sign.c文件编译成.o文件
2. 通过ld命令将刚刚生成的文件链接成bin/bootblock文件，该文件是一个目前lab1的bootloader 如下所示

```
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes/lab1/bin$ file
bootblock
```

```
bootblock: DOS/MBR boot sector
```

123

1. 通过dd命令将上方生成的bootblock 和 kernel 生成操作系统镜像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？

这部分我其实没有研究出来，查了别人的博客才发现说是bootloader的验证(特征)是在sign.c中的

原文链接 [https://blog.csdn.net/qq\\_41946412/article/details/103091369](https://blog.csdn.net/qq_41946412/article/details/103091369)

123

```
char buf[512];
memset(buf, 0, sizeof(buf));
FILE *ifp = fopen(argv[1], "rb");
int size = fread(buf, 1, st.st_size, ifp);
if (size != st.st_size) {
    fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
    return -1;
}
fclose(ifp);
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;
}
fclose(ofp);
printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
return 0;
1234567891011121314151617181920
```

符合规范的硬盘主引导扇区的特征应该包括，

- ①大小为512个字节，
- ②没用到的其他位置设置为0，
- ③第511个字节0x55，第512个字节是0xAA，也就是说，最后一个和倒数第二个字节的内容是确定的，
- ④如果读出的字节数不是512，需要报错。

即应该可以读出512.

## 练习二

### 题目

使用qemu执行并调试lab1中的软件。（要求在报告中简要写出练习过程）

为了熟悉使用qemu和gdb进行的调试工作，我们进行如下的小练习：

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。
2. 在初始化位置0x7c00设置实地址断点,测试断点正常。
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。
4. 自己找一个bootloader或内核中的代码位置，设置断点并进行测试。

## 解答

```
$ cd labcodes_answer/lab1_result/  
$ make lab1-mon 在命令行中输入该命令即可进入调试模式  
12
```

查看[makefile](#)文件中该命令的所在地

```
lab1-mon: $(UCOREIMG)  
    $(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -monitor  
stdio -hda $< -serial null"  
    $(V)sleep 2  
    $(V)$(TERMINAL) -e "gdb -q -x tools/lab1init"  
1234
```

通过上面的代码我们可以看出来，这个代码大致是干了两件事

1. 是把qemu 执行的指令记录下来放在 q.log中
2. 和我们的gdb结合来调试我们的bootloader,调试指令放在tools/lab1init中，下面我们来看一下这些指令

```
# 这些都是gdb可以识别的命令  
file bin/kernel # 这条命令是在加载bin/kernel  
target remote :1234 # 与qemu进行连接  
set architecture i8086 # 设置cpu结构为 i8086 (我的理解)  
b *0x7c00 # 这个是设置断点 # 设置bootloader启动的断点  
continue # 然后让系统继续运行  
x /2i $pc # 打印相应的指令 i是指令的意思 /2i 的意思即是打印两条指令  
1234567
```

通过课程的知识我们可以知道，当cpu加电启动bios后，等到bios进入0x7c00处，开始运行bootloader的代码，把控制权交给bootloader

在命令行输入make lab1-mon后，弹出来几个窗口，其中有qemu窗口以及gdb调试窗口

其中qdb调试窗口显示如下

```
0x0000fff0 in ?? ()  
The target architecture is assumed to be i8086  
Breakpoint 1 at 0x7c00  
  
Breakpoint 1, 0x00007c00 in ?? ()  
=> 0x7c00: c1i  
    0x7c01: c1d  
(gdb)  
12345678
```

输入 x /10i \$pc 指令，我们就可以打印出pc指令寄存器之后的10条指令

如下

```
(gdb) x /10i $pc
=> 0x7c00: cli
    0x7c01: cld
    0x7c02: xor    %eax,%eax
    0x7c04: mov    %eax,%ds
    0x7c06: mov    %eax,%es
    0x7c08: mov    %eax,%ss
    0x7c0a: in     $0x64,%al
    0x7c0c: test   $0x2,%al
    0x7c0e: jne    0x7c0a
    0x7c10: mov    $0xd1,%al
1234567891011
```

而这些指令正好与我们boot/bootasm.S文件中第16行开始是对应的  
bootasm.S文件中的代码如下

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

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

    # Enable A20:
    # For backwards compatibility with the earliest PCs, physical
    # address line 20 is tied low, so that addresses higher than
    # 1MB wrap around to zero by default. This code undoes this.
1234567891011121314151617
```

1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

1

1. 在终端进入os\_kernel\_lab/labcodes\_answer/lab1\_result

2. 修改./tools/gdbinit如下

```
set architecture i8086
target remote :1234
12
```

首先使用make clean，再make debug进行跟踪调试

ps:[如果你的cgdb调试时出现了[34m](#)等颜色代码，[可以看这篇文章](#)

回想上课说的cpu加电第一条指令是在cs:ip的位置，即cs寄存器的值左移四位加上eip寄存器的值

随即我们就可以打开cgdb调试页面

```
leezed@leezed-Ubuntu: ~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result
QEMU (Paused)
loader/
CGDB: a curses debugger
version 0.6.7

type q<Enter>      to exit
type help<Enter>   for GDB help
type <ESC>:help<Enter> for CGDB help

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
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb)
drwxrwxr-x 6 leezed leezed 4096 5月 12 12:19 obj/
-rw-rw-r-- 1 leezed leezed 0 5月 3 00:11 .projectile
-rw-rw-r-- 1 leezed leezed 16533 5月 3 00:11 report.md
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result$ make debug
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
# 选项 "-e" 已弃用并可能在 gnome-terminal 的后续版本中移除。
# 使用 "--" 以结束选项并将要执行的命令行追加至其后。
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result$
```

我们查看 cs寄存器及eip寄存器的地址

```
CGDB: a curses debugger
version 0.6.7

type q<Enter>      to exit
type help<Enter>   for GDB help
type <ESC>:help<Enter> for CGDB help

(gdb) x /2i 0x0000ffff0
0xfffff0:    jmp    $0x3630,$0xf000e05b
0xfffff7:    das
(gdb) x /2i 0x0000ffff0
=> 0xfffff0:    add    %al, (%eax)
0xfffff2:    add    %al, (%eax)
(gdb) p/x $cs
$1 = 0xf000
(gdb) p/x $eip
$2 = 0xfffff0
(gdb)
```

我们手动进行评价，  
cs左移四位及0xf0000  
加上eip  
即0xfffff0  
我们查看cs:ip 的指令



```
终端

CGDB: a curses debugger
version 0.6.7

type q<Enter>          to exit
type help<Enter>       for GDB help
type <ESC>:help<Enter> for CGDB help

(gdb) x /2i 0x0000fff0
=> 0xffff0:      add    %al, (%eax)
    0xffff2:      add    %al, (%eax)
(gdb) p/x $cs
$1 = 0xf000
(gdb) p/x $eip
$2 = 0xffff0
(gdb) x /2i 0xfffff0
    0xfffff0:      ljmp   $0x3630, $0xf000e05b
    0xfffff7:      das
(gdb) 
= 10
```

可以看到是一条长跳转指令

之后进行单步调试，具体的就不调试了。

2. 在初始化位置0x7c00设置实地址断点，测试断点正常。
- 1

修改tools/gdbinit文件设置断点

```
file obj/bootblock.o
set architecture i8086
target remote :1234
b *0x7c00
continue
12345
```

这时候我们可以看到cgdb调试页面就是  
boot/bootasm.S的入口代码

```

leezed@leezed-Ubuntu: ~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result
QEMU [Paused]
ffff0000 0x00100043

14: start:
15: .code16 # Assemble for 16-bit mo
16: cli # Disable interrupts
17: cld # String operations incr
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:
> leezed/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result/boot/bootasm.S
The target architecture is assumed to be i8086
0x0000ffff in ?? ()
Breakpoint 1 at 0x7c00: file boot/bootasm.S, line 16.
1: ds
1: ss
Breakpoint 1, start () at boot/bootasm.S:16
+++ swi cli # Disable interrupts
2: @rin (qdb) p/x %cs
2: cs $1 = 0x0
2: ds (qdb) p/x %eip
2: ss $2 = 0x7c00
100 tic (qdb)
^C
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result$ gedit Makefile
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result$ gedit tools/gdbinit
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result$ make debug
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
# 选项 "-e" 已弃用并可能在 gnome-terminal 的后续版本中移除。
# 使用 "--" 以结束选项并将要执行的命令行追加至其后。
leezed@leezed-Ubuntu:~/projects/ucore/os_kernel_lab/labcodes_answer/lab1_result$
  
```

[查看bootasm.S文件](#)[illegible]

```

# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al                # wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al               # 0xd1 -> port 0x64
    outb %al, $0x64              # 0xd1 means: write data to
8042's P2 port

seta20.2:
    inb $0x64, %al                # wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al              # 0xdf -> port 0x60
    outb %al, $0x60              # 0xdf = 11011111, means set
P2's A20 bit(the 1 bit) to 1

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

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

.code32                          # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    movw $PROT_MODE_DSEG, %ax    # Our data segment selector
    movw %ax, %ds                # -> DS: Data Segment
    movw %ax, %es                # -> ES: Extra Segment
    movw %ax, %fs                # -> FS
    movw %ax, %gs                # -> GS
    movw %ax, %ss                # -> SS: Stack Segment

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

    # If bootmain returns (it shouldn't), loop.
spin:
    jmp spin

```

```

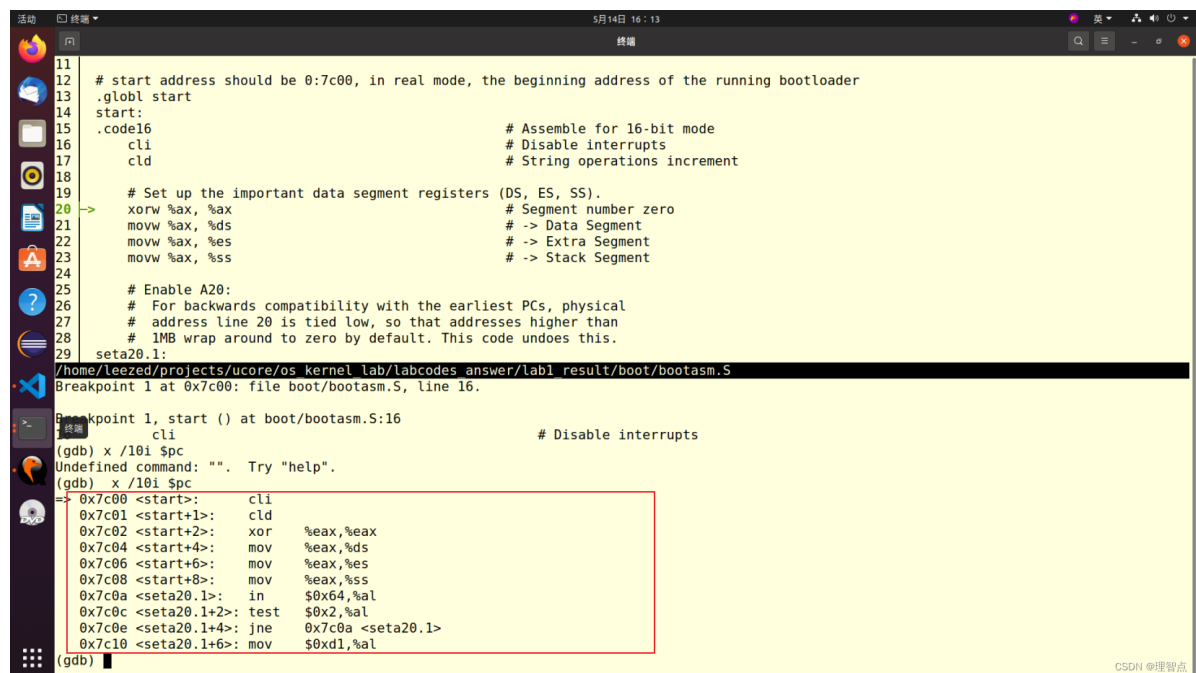
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
    SEG_NULLASM                            # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg for bootloader and
kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)       # data seg for bootloader and
kernel

gdt desc:
    .word 0x17                             # sizeof(gdt) - 1
    .long gdt                             # address gdt

12345678910111213141516171819202122232425262728293031323334353637383940414243444
54647484950515253545556575859606162636465666768697071727374757677787980818283848
58687

```

反汇编的得到的运行命令为



```

11
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:
/home/leezed/projects/ucore/os kernel lab/labcodes/answer/lab1 result/boot/bootasm.S
Breakpoint 1 at 0x7c00: file boot/bootasm.S, line 16.
Breakpoint 1, start () at boot/bootasm.S:16
    cli                                    # Disable interrupts
(gdb) x /10i $pc
Undefined command: "". Try "help".
(gdb) x /10i $pc
0x7c00 <start>:  cli
0x7c01 <start+1>:  cld
0x7c02 <start+2>:  xor    %eax,%eax
0x7c04 <start+4>:  movw   %eax,%ds
0x7c06 <start+6>:  movw   %eax,%es
0x7c08 <start+8>:  movw   %eax,%ss
0x7c0a <seta20.1>: in     $0x64,%al
0x7c0c <seta20.1+2>: test  $0x2,%al
0x7c0e <seta20.1+4>: jne   0x7c0a <seta20.1>
0x7c10 <seta20.1+6>: mov   $0xd1,%al
(gdb)

```

可以看到这里的代码与bootasm.S中的文件大同小异

4. 自己找一个bootloader或内核中的代码位置，设置断点并进行测试。

1

略

## 练习三：分析bootloader进入保护模式的过程。（要求在报告中写出分析）

# 题目

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。请分析bootloader是如何完成从实模式进入保护模式的。

提示：需要阅读小节“保护模式和分段机制”和lab1/boot/bootasm.S源码，了解如何从实模式切换到保护模式，需要了解：

1. 为何开启A20，以及如何开启A20
2. 如何初始化GDT表
3. 如何使能和进入保护模式

## 解答

1. 为何开启A20，以及如何开启A20

### 为何开启A20

首先在bootloader接手bios的工作时，pc系统仍然处于实模式（16位模式）下，这时候只能使用20根地址总线，在这种状态下软件可以访问的物理内存空间不超过1MB,就无法发挥i80386 32位cpu的4GB内存管理能力

具体来说，这是一个历史性问题。在intel处理器8086中，“段：偏移”最大能表示的内存地址是FFFF:FFFF，即10FFEFh，但是8086仅仅有20位寻址地址总线，仅仅能寻址到1MB，假设试图访问1MB以上的内存地址，并不会错误发生，而是回卷。即又回到0000:0000地址，又从零开始寻址。但是到了80286时，真的能够访问到1MB以上的内存了。假设遇到相同的情况，系统不会再回卷寻址，这就造成了向上不兼容，为了保证100%兼容，IBM想出了一个办法。使用8042键盘控制器来控制第20个地址位。这就是A20地址线。

### 如何开启A20

可以查看boot/bootasm.S文件中的这段代码

```
# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al                                # wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al                                # 0xd1 -> port 0x64
    outb %al, $0x64                                # 0xd1 means: write data to
8042's P2 port

seta20.2:
    inb $0x64, %al                                # wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al                                # 0xdf -> port 0x60
```

```
outb %a1, $0x60                                # 0xdf = 11011111, means set
P2's A20 bit(the 1 bit) to 1
12345678910111213141516171819
```

打开A20方法，不断查看0x64端口的值，如果是2表示不忙，如果不忙，把0xd1写入0x64端口，把0xdf写入0x60端口

## 如何初始化GDT表

初始化gdt表通过bootams.S文件中的这条命令来实现

```
lgdt gdt desc
1
```

那么这条命令干了什么呢，我们先来看看gdt desc是啥

```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
    SEG_NULLASM                            # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg for bootloader and
kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)       # data seg for bootloader and
kernel

gdt desc:
    .word 0x17                            # sizeof(gdt) - 1
    .long gdt                             # address gdt

1234567891011
```

因没有精力完成该项目，现进行搁置

## 小结

### bios启动过程

以Intel 80386为例，计算机加电后，CPU从物理地址0xFFFFF0（由初始化的CS: EIP确定，此时CS和IP的值分别是0xF000和0xFFFF0）开始执行。在0xFFFFF0这里只是存放了一条跳转指令，通过跳转指令跳到BIOS例行程序起始点。BIOS做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区（即主引导扇区或启动扇区）到内存一个特定的地址0x7c00处，然后CPU控制权会转移到那个地址继续执行。至此BIOS的初始化工作做完了，进一步的工作交给了ucore的bootloader。

### bootloader启动过程

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。bootloader完成的工作包括：

1. 切换到保护模式，启用分段机制
2. 读磁盘中ELF执行文件格式的ucore操作系统到内存
3. 显示字符串信息
4. 把控制权交给ucore操作系统

对应其工作的实现文件在lab1中的boot目录下的三个文件asm.h、bootasm.S和bootmain.c。