
操作系统课程设计实验报告

实验名称： 教学操作系统 ucore 实验 1: OS 启动、设备与中断管理

姓名/学号： 刘鑫/1120181208

一、 实验目的

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-**bootloader** 来完成这些工作。为此，我们需要完成一个能够切换到 x86 的保护模式并显示字符的 **bootloader**，为启动操作系统 **ucore** 做准备。**lab1** 提供了一个非常小的 **bootloader** 和 **ucore OS**，整个 **bootloader** 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个 **bootloader** 和 **ucore OS**，同学们可以了解到：

- 计算机原理
 - CPU 的编址与寻址：基于分段机制的内存管理
 - CPU 的中断机制
 - 外设：串口/并口/CGA，时钟，硬盘
- Bootloader 软件
 - 编译运行 **bootloader** 的过程
 - 调试 **bootloader** 的方法
 - PC 启动 **bootloader** 的过程
 - ELF 执行文件的格式和加载
 - 外设访问：读硬盘，在 CGA 上显示字符串
- ucore OS 软件
 - 编译运行 **ucore OS** 的过程
 - **ucore OS** 的启动过程
 - 调试 **ucore OS** 的方法
 - 函数调用关系：在汇编级了解函数调用栈的结构和处理过程
 - 中断管理：与软件相关的中断处理
 - 外设管理：时钟

二、 实验内容

练习 1：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：

-
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？

练习 2：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。

练习 3：分析 `bootloader` 进入保护模式的过程。（要求在报告中写出分析）

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。

练习 4：分析 `bootloader` 加载 ELF 格式的 OS 的过程。（要求在报告中写出分析）

通过阅读 `bootmain.c`，了解 `bootloader` 如何加载 ELF 文件。通过分析源代码和通过 `qemu` 来运行并调试 `bootloader&OS`，

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

练习 5：实现函数调用堆栈跟踪函数（需要编程）

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：

```

.....
ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094 0x00007b58
0x00100096

    kern/debug/kdebug.c:305: print_stackframe+22

ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000
0x00007ba8

    kern/debug/kmonitor.c:125: mon_backtrace+10

ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000
0x00007b84

    kern/init/init.c:48: grade_backtrace2+33

ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4
0x00000029

    kern/init/init.c:53: grade_backtrace1+38

ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000
0x0000001d

    kern/init/init.c:58: grade_backtrace0+23

ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308
0x00000000

    kern/init/init.c:63: grade_backtrace+34

ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000
0x00007c53

    kern/init/init.c:28: kern_init+88

ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e
0xfa7502a8

<unknow>: -- 0x00007d72 -
.....

```

请完成实验，看看输出是否与上述显示大致一致，并解释最后一行各个数值的含义。

提示：可阅读小节“函数堆栈”，了解编译器如何建立函数调用关系的。在完成 lab1 编译后，查看 lab1/obj/bootblock.asm，了解 bootloader 源码与机器码的语句和地址等的对应关系；查看 lab1/obj/kernel.asm，了解 ucore OS 源码与机器码的语句和地址等的对应关系。

要求完成函数 `kern/debug/kdebug.c::print_stackframe` 的实现，提交改进后源代码包（可以编译执行），并在实验报告中简要说明实现过程，并写出对上述问题的回答。

补充材料：

由于显示完整的栈结构需要解析内核文件中的调试符号，较为复杂和繁琐。代码中有一些辅助函数可以使用。例如可以通过调用 `print_debuginfo` 函数完成查找对应函数名并打印至屏幕的功能。具体可以参见 `kdebug.c` 代码中的注释。

练习 6：完善中断初始化和处理（需要编程）

请完成编码工作和回答如下问题：

1. 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？
2. 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`。在 `idt_init` 函数中，依次对所有中断入口进行初始化。使用 `mmu.h` 中的 `SETGATE` 宏，填充 `idt` 数组内容。每个中断的入口由 `tools/vectors.c` 生成，使用 `trap.c` 中声明的 `vectors` 数组即可。
3. 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”。

三、 实验环境

Windows:

Windows10

处理器 Inter core i5-8265u@1.60GHz

内存 8.00GB

系统类型 64 位操作系统，基于 x64 的处理器

Linux:

虚拟机软件：Oracle VM VitrualBox

虚拟机操作系统: ubuntu 14.04 x86-64

虚拟机内存: 4GB

虚拟机硬盘容量:60GB

四、 程序设计与实现

练习一

问题一:

操作步骤:

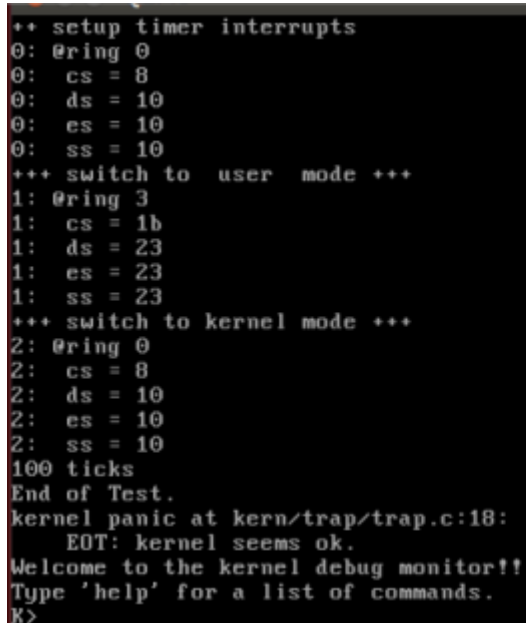
一 执行 make qemu

1. 进入/moocos/ucore_lab/labcodes_answer/lab1_result 目录

cd /moocos/ucore_lab/labcodes_answer/lab1_result

2. 执行 make qemu

make qemu



```
++ setup timer interrupts
0: 0ring 0
0: cs = 8
0: ds = 10
0: es = 10
0: ss = 10
+++ switch to user mode +++
1: 0ring 3
1: cs = 1b
1: ds = 23
1: es = 23
1: ss = 23
+++ switch to kernel mode +++
2: 0ring 0
2: cs = 8
2: ds = 10
2: es = 10
2: ss = 10
100 ticks
End of Test.
kernel panic at kern/trap/trap.c:18:
  EOT: kernel seems ok.
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

ctrl+c 退出

二 Makefile 文件分析

相关知识:

在软件开发中, make 通常被视为一种软件构建工具。该工具主要经由读取一种名为“makefile”或“Makefile”的文件来实现软件的自动化建构。它会通过一种被称之为“target”概念来检查相关文件之间的依赖关系,这种依赖关系的检查系统非常简单,主要通过对比文件的修改时间来实现。在大多数情况下,我们主要用它来编译源代码,生成结果代码,然后把结果代码连接起来生成可执行文件或者库文件。

生成 ucore.image 的代码:

```
# 生成ucore.image的代码
UCOREIMG := $(call totarget,ucore.img) #创建了一个新的totarget函数
#call 函数是唯一一个可以用来创建新的参数化的函数。我们可以用来写一个非常复杂的表达式,
#这个表达式中,我们可以定义很多的参数,然后你可以用 call 函数来向这个表达式传递参数。

$(UCOREIMG): $(kernel) $(bootblock) #完成make qemu的命令需要生成kernel和bootblock文件
#$(kernel) $(call totarget,kernel)
#$(bootblock) $(call totarget,bootblock)

#生成ucore.image首先需要生成bootblock和kernel
#dd可从标准输入或文件中读取数据,根据指定的格式来转换数据,再输出到文件、设备或标准输出。
#参数说明:
#if=文件名: 输入文件名,默认为标准输入。即指定源文件。
#of=文件名: 输出文件名,默认为标准输出。即指定目的文件。
#count=blocks: 仅拷贝blocks个块,块大小等于ibs指定的字节数。
#conv=<关键字>, noerror: 出错时不停止,
#seek=blocks: 从输出文件开头跳过blocks个块后再开始复制。
$(V)dd if=/dev/zero of=$@ count=10000 #创建10000块扇区,每个扇区512字节,制成ucore.img虚拟磁盘
$(V)dd if=$(bootblock) of=$@ conv=notrunc #将bootblock存到ucore.img虚拟磁盘的第一块
$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc #将kernel存到ucore.img虚拟磁盘的第二块及之后几块,注意seek1,最终ucore.img虚拟磁盘制作完成

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

首先需要生成 bootblock 和 kernel

生成 bootblock 的相关代码:

```
# 生成bootblock的相关代码
bootfiles = $(call listf_cc,boot)

#$(foreach <var>,<list>,<text>)
#函数的功能是:把参数<list>中的单词逐一取出放到参数<var>所指定的变量中,
#然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串,循环过程中,
#<text>的返回的每个字符串会以空格分割,最后当整个循环结束的时候,<text>
#所返回的每个字符串所组成的整个字符串(以空格分隔)将会是 foreach 函数的返回值。
#所以<var>最好是一个变量名,<list>可以是一个表达式,而<text>中一般会只用<var>这个参数来一次枚举<list>中的单词。
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc)) #生成bootasm.o,bootmain.o

bootblock = $(call totarget,bootblock) #链接所有的.o文件

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign) #生成bootblock需要bootasm.o、bootmain.o、sign工具
@echo + ld $@
$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock) #将文件与库连接为可执行文件
@$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock) #OBJDUMP := $(GCCPREFIX)objdump 查看目标文件或者可执行文件的构成
@$(OBJDUMP) -t $(call objfile,bootblock) | $(SED) '1,/SYMBOL TABLE/d;s/ / /; /<$/d' > $(call symfile,bootblock) #生成bootblock.o文件
@$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock) #将生成的bootblock.o文件的内容拷贝到bootblock.out中,并统一为二进制文件格式
@$(call totarget,sign) $(call outfile,bootblock) $(bootblock) #使用sign工具处理bootblock.out文件,生成bootblock

$(call create_target,bootblock)
```

为了生成 bootblock, 首先需要生成 bootasm.o、bootmai.o、sign 工具

生成 sign 工具的代码如下：

```
"  
# 生成sign工具部分的代码  
$(call add_files_host,tools/sign.c,sign,sign) #编译tools/sign.c, 生成sign.o  
$(call create_target_host,sign,sign) #将sign.o链接为sign  
# -----
```

执行的实际命令为：

```
+ cc tools/sign.c  
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
```

生成 bootasm.o 和 bootmain.o 部分的代码：

```
bootfiles = $(call listf_cc,boot)  
  
#$(foreach <var>,<list>,<text>)  
#函数的功能是：把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，  
#然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，  
#<text>的返所返回的每个字符串会以空格分割，最后当整个循环结束的时候，<text>  
#所返回的每个字符串所组成的整个字符串（以空格分隔）将会是 foreach 函数的返回值。  
#所以<var>最好是一个变量名，<list>可以是一个表达式，而<text>中一般会只用<var>这个参数来一次枚举<list>中的单词。  
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc) #生成bootasm.o,bootmain.o
```

生成 bootasm.o 需要 bootasm.S

执行的实际命令为：

```
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  
-fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o  
obj/boot/bootasm.o
```

其中关键的参数为：

-ggdb 生成可供 gdb 使用的调试信息。这样才能用 qemu+gdb 来调试 bootloader or ucore。

-m32 生成适用于 32 位环境的代码。我们用的模拟硬件是 32bit 的 80386，所以 ucore 也要是 32 位的软件。

-gstabs 生成 stabs 格式的调试信息。这样要 ucore 的 monitor 可以显示出便于开发者阅读的函数调用栈信息

-nostdinc 不使用标准库。标准库是给应用程序用的，我们是编译 ucore 内核，OS 内核是提供服务的，所以所有的服务要自给自足。

`-fno-stack-protector` 不生成用于检测缓冲区溢出的代码。这是 for 应用程序的，我们是编译内核，ucore 内核好像还用不到此功能。

`-Os` 为减小代码大小而进行优化。根据硬件 spec，主引导扇区只有 512 字节，我们写的简单 bootloader 的最终大小不能大于 510 字节。

`-I<dir>` 添加搜索头文件的路径

生成 bootmain.o 需要 bootmain.c

执行的实际命令为：

```
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  
-fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o  
obj/boot/bootmain.o
```

新出现的关键参数：

`-fno-builtin` 除非用 `__builtin_` 前缀，否则不进行 builtin 函数的优化

首先生成 bootblock.o

实际执行的命令：

```
ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o  
obj/boot/bootmain.o -o obj/bootblock.o
```

其中关键的参数为

`-m <emulation>` 模拟为 i386 上的连接器

`-nostdlib` 不使用标准库

`-N` 设置代码段和数据段均可读写

`-e <entry>` 指定入口

`-Ttext` 制定代码段开始位置

拷贝二进制代码 bootblock.o 到 bootblock.out

实际执行的指令：

```
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 <bfdname> 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

生成 kernel 的代码部分

```
$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS)) #用于生成.o文件, 生成的.o文件用于生成kernel

KOBJS = $(call read_packet,kernel libs)

# 生成kernel的相关代码
kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld #kernel.ld是已经存在的文件

$(kernel): $(KOBJS)
    @echo + ld $@
    $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
#将
#kernel.ld init.o readline.o stdio.o kdebug.o
# kmonitor.o panic.o clock.o console.o intr.o picirq.o trap.o
# trapentry.o vectors.o pmm.o printfmt.o string.o
#多个目标文件链接为可执行文件, 使用指定的tools/kernel.ld脚本文件来进行链接

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

$(call create_target,kernel)
```

生成 kernel, 需要两步

1. 编译 kern/目录下的 C 程序, 生成 kernel 需要的.o 文件

```
$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))
```

2. 链接这些.o 文件, 生成 kernel

执行的实际命令为:

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/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
```

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

生成一个有 10000 个块的文件，每个块默认 512 字节，用 0 填充。

执行的实际命令为：

```
dd if=/dev/zero of=bin/ucore.img count=10000
```

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

将 bootblock 存到 ucore.img 虚拟磁盘的第一块

执行的实际命令为：

```
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
```

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

将 kernel 存到 ucore.img 虚拟磁盘的第二块及之后几块，注意 seek1，最终 ucore.img 虚拟磁盘制作完成

执行的实际命令为：

```
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
```

问题二：

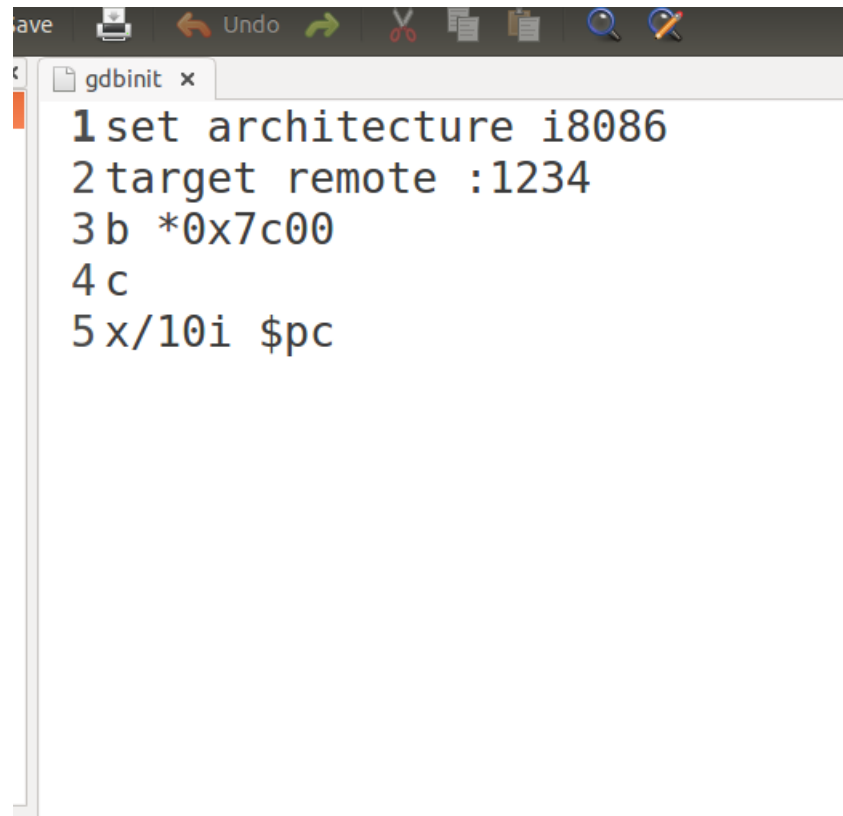
引导扇区的大小为 512 字节，最后两个字节为标志性结束字节 0x55, 0xAA，做完这样的检查才能认为是符合规范的磁盘主引导扇区。

练习二

练习 2.1

1. 修改 gdbinit 文件

首先，在 `/moocos/ucore_lab/labcodes_answer/lab1_result/tools` 目录下，修改 `gdbinit` 文件



The screenshot shows a text editor window with a dark theme. The title bar at the top includes icons for save, print, undo, redo, cut, copy, paste, search, and a pencil. The editor has a single tab labeled 'gdbinit x'. The text inside the editor is as follows:

```
1 set architecture i8086
2 target remote :1234
3 b *0x7c00
4 c
5 x/10i $pc
```

2. make debug

输入 `cd ..` 返回上级目录，然后输入 `make debug`，弹出窗口如图：

```
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

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) █
```

3. 在 gdb 窗口中使用 si 命令便可以实现单步追踪

```
The target architecture is assumed to be i8086
0x0000fff0 in ?? ()
(gdb) si
0x0000e05b in ?? ()
(gdb)
0x0000e062 in ?? ()
(gdb)
0x0000e066 in ?? ()
(gdb) █
```

4. 在 gdb 界面下，可通过如下命令来看 BIOS 的代码 x /2i \$pc（显示当前 eip 处的汇编指令）

```

0x7c0a:    in    $0x64,%al
0x7c0c:    test  $0x2,%al
0x7c0e:    jne   0x7c0a
0x7c10:    mov   $0xd1,%al
(gdb) x/2i $pc
=> 0x7c00:    cli
    0x7c01:    cld
(gdb)

```

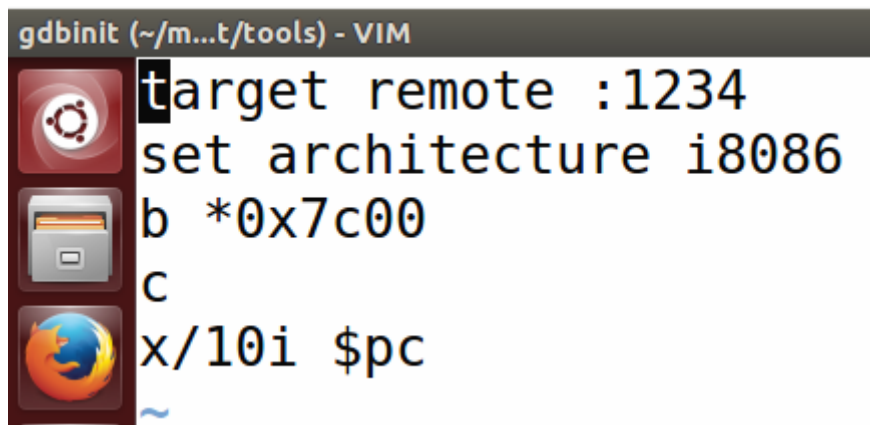
练习 2.2

1. 修改 `gdbinit` 文件

进入目录：

```
cd ./moocos/ucore_lab/labcodes_answer/lab1_result/tools
```

修改内容如下：



```

gdbinit (~/.m...t/tools) - VIM
target remote :1234
set architecture i8086
b *0x7c00
c
x/10i $pc
~

```

2. `make debug`

返回上级目录，输入 `make debug`

```
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

Breakpoint 1, 0x00007c00 in ?? ()
=> 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) █
```

练习 2.3

反汇编得到的代码：

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

Breakpoint 1, 0x00007c00 in ?? ()
=> 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) █
```

bootasm.S 中部分代码

```

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.

```

bootblock.asm 中部分的代码：

```

10 .globl start
11 start:
12 .code16
    # Assemble for 16-bit mode
13
    cli
    # Disable interrupts
14     7c00:
    fa
    cli
15 |
    cld 英 C 09 ⚙️
    # String operations increment

```

```

16      7c01: '
      fc
      cld
17
18      # Set up the important
      data segment registers (DS,
      ES, SS).
19      xorw %ax, %
      ax
      # Segment number zero
20      7c02:      31
      c0
      xor      %eax,%eax

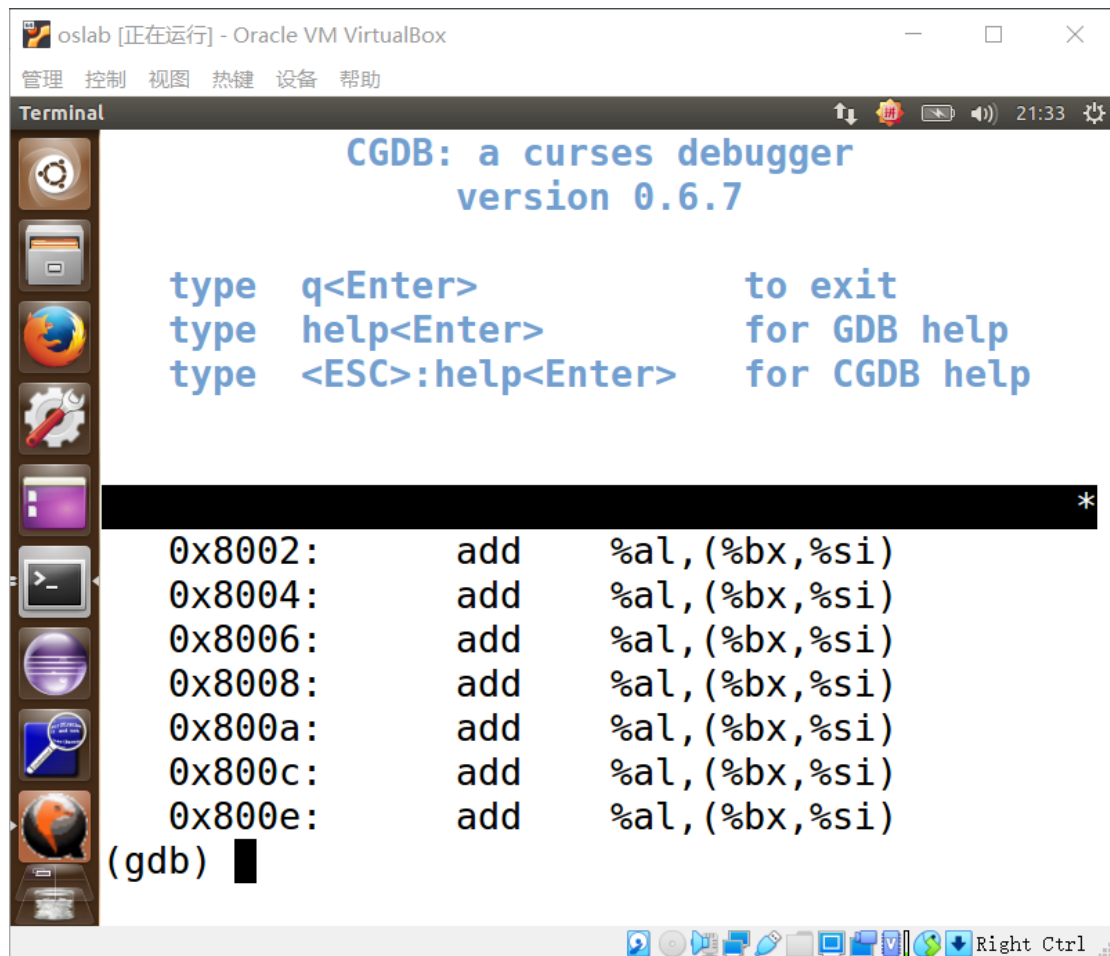
20      7c02:      31
      c0
      xor      %eax,%eax
21      movw %ax, %
      ds
      # -> Data Segment
22      7c04:      8e
      d8
      mov      %eax,%ds
23      movw %ax, %
      es
      # -> Extra Segment
24      7c06:      8e
      c0

```

三者对比基本一致

练习 2.4

设置断点地址为 0x8002，开始测试



练习三

(一) 代码分析

```
#include <defs.h>
#include <x86.h>
#include <elf.h>
/*
    bootasm.S 完成了 bootloader 的大部分功能
    包括打开 A20, 初始化 GDT, 进入保护模式, 更新段寄存器的
    值, 建立堆栈
*/

/*
*****
* 这是一个非常简单的引导加载程序, 它的唯一工作
就是引导
* 来自第一个 IDE 硬盘的 ELF 内核映像
*****
```

```

*
* 磁盘布局
* 这个程序 (bootasm)。S 和 bootmain.c) 是引导加载
程序。
* 应该存储在磁盘的第一个扇区。
*
** 第二个扇区包含内核映像。
*
** 内核映像必须是 ELF 格式。
*
* 开机步骤
** 当 CPU 启动时，它将 BIOS 加载到内存中并执行它
*
** BIOS 初始化设备，设置中断例程，以及
* 读取启动设备（硬盘）的第一个扇区
* 进入内存并跳转到它。
*
** Assuming this boot loader is stored in the
first sector of the
* hard - drive, this code takes over...
*
** 控制启动 bootasm.S -- 设置保护模式，
* 和一个堆栈，C 代码然后运行，然后调用 bootmain()
*
** bootmain() 在这个文件中接管，读取内核并跳转到
它
**/
unsigned int    SECTSIZE =    512 ;
struct elfhdr * ELFHDR   =    ((struct elfhdr
*)0x10000) ;    // scratch space

/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}

/* readsect - read a single sector at @secno into @dst
*/
//读取扇区的代码如下：
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready

```

```

        waitdisk();

        outb(0x1F2, 1);                                //
count = 1 outb(使用内联汇编实现), 设置读取扇区的数目为
1
        outb(0x1F3, secno & 0xFF);
        outb(0x1F4, (secno >> 8) & 0xFF);
        outb(0x1F5, (secno >> 16) & 0xFF);
        outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
        //上面四条指令联合制定了扇区号在这 4 个字节联合
        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
        outb(0x1F7, 0x20);                            // 0x20
命令, 读取扇区
        // wait for disk to be ready
        waitdisk();
        //将扇区内容加载到内存中虚拟地址 dst
        // read a sector
        insl(0x1F0, dst, SECTSIZE / 4);
//也用内联汇编实现

```

```

    }

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);    //读到 dst
位置, 幻数 (直接使用的常数叫做幻数) 4 因为这里以 dw 为单
位
    }

```

//从 outb() 可以看出这里是用 LBA 模式的 PIO(Program IO) 方式来访问硬盘的 (即所有的 IO 操作是通过 CPU 访问硬盘的 IO 地址寄存器完成), 该函数一次只读一个扇区

```

/*
读取硬盘扇区的步骤:

```

1. 等待硬盘空闲。waitdisk 的函数实现只有一行：while ((inb(0x1F7) & 0xC0) != 0x40)，意思是不断查询读 0x1F7 寄存器的最高两位，

直到最高位为 0、次高位为 1（这个状态应该意味着磁盘空闲）才返回。

2. 硬盘空闲后，发出读取扇区的命令。对应的命令字为 0x20，放在 0x1F7 寄存器中；读取的扇区数为 1，放在 0x1F2 寄存器中；

读取的扇区起始编号共 28 位，分成 4 部分依次放在 0x1F3~0x1F6 寄存器中。

3. 发出命令后，再次等待硬盘空闲。

4. 硬盘再次空闲后，开始从 0x1F0 寄存器中读数据。注意 insl 的作用是

“That function will read cnt dwords from the input port specified by port into the supplied output array addr.”，

是以 dword 即 4 字节为单位的，因此这里 SECTIZE 需要除以 4。

```
    */
/* *
 * readseg - read @count bytes at @offset from kernel
into virtual address @va,
 * might copy more than asked.
 */
static void
readseg(uintptr_t va, uint32_t count, uint32_t
offset) {
    uintptr_t end_va = va + count;

    // round down to sector boundary
    va -= offset % SECTSIZE;

    // translate from bytes to sectors; kernel starts
at sector 1
    uint32_t secno = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of
sectors at a time.
```

```

        // We'd write more to memory than asked, but it
        doesn't matter --
        // we load in increasing order.
        for (; va < end_va; va += SECTSIZE, secno++) {
            readsect((void *)va, secno);
        }
    }

    /* bootmain - the entry of bootloader */
    void
    bootmain(void) {
        // read the 1st page off disk
        //读取 ELF 的头部
        readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

        // is this a valid ELF?
        //判断是否是合法的 ELF 文件
        if (ELFHDR->e_magic != ELF_MAGIC) {
            goto bad;
        }

        struct proghdr *ph, *eph;

        // load each program segment (ignores ph flags)
        // ELF 头部有描述 ELF 文件应加载到内存什么位置的
        // 描述表,
        // 先将描述表的头地址存在 ph
        ph = (struct proghdr *)((uintptr_t)ELFHDR +
            ELFHDR->e_phoff);
        eph = ph + ELFHDR->e_phnum;
        //按照描述表将 ELF 文件中数据载入缓存
        for (; ph < eph; ph++) {
            readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz,
                ph->p_offset);
        }

        // call the entry point from the ELF header
        // note: does not return
        // ELF 文件 0x1000 位置后面的 0xd1ec 比特被载入
        // 内存 0x00100000
        // ELF 文件 0xf000 位置后面的 0x1d20 比特被载入内
        // 存 0x0010e000
        // 根据 ELF 头部储存的入口信息, 找到内核的入口

```

```

        ((void (*)(void))(ELFHDR->e_entry &
0xFFFFFFFF)) ();
        //跳到内核程序入口地址，将 cpu 控制权交给 ucore
内核代码
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}
/*
* bootloader 是如何加载 ELF 格式的 OS?
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；
    首先看 readsect 函数，readsect 从设备的第 secno 扇区
读取数据到 dst 位置
    readseg 简单包装了 readsect，可以从设备读取任意长度
的内容。
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。
*/

```

（二） 问题解答

1.1 为何开启 A20，以及如何开启 A20

1981 年 8 月，IBM 公司最初推出的个人计算机 IBM PC 使用的 CPU 是 Inter 8088. 在该微机中地址线只有 20 根。在当时内存 RAM 只有几百 KB 或不到 1MB 时，20 根地址线已经足够用来寻址这些 内存。其所能寻址的最高地址是 0xffff，

也就是 0x10ffef。对于超出 0x100000 (1MB) 的寻址地址将默认地环绕到 0xffef。当 IBM 公司与 1985 年引入 AT 机时，使用的是 Inter 80286 CPU，具有 24 根地址线，最高可寻址 16MB，并且有一个与 8088 那样实现地址寻址的环绕。

但是当时已经有一些程序是利用这种环绕机制进行工作的。为了实现完全的兼容性，IBM 公司发明了使用一个开关来开启或禁止 0x100000 地址比特位。由于当时的 8042 键盘控制器上恰好有空闲的端口引脚（输出端口 P2，引脚 P21），

于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为 A20。如果它为零，则比特 20 及以上地址都被清除。从而实现了兼容性。

当 A20 地址线控制禁止时，程序就像运行在 8086 上，1MB 以上的地址是不可访问的，只能访问奇数 MB 的不连续的地址。为了使能所有地址位的寻址能力，必须向键盘控制器 8082 发送一个命令，键盘控制器 8042 会将 A20 线置于高电位，使全部 32 条地址线可用，实现访问 4GB 内存。

1.2 打开 A20 的具体步骤

由于在机器启动时，默认条件下，A20 地址线是禁止的，所以操作系统必须使用适当的方法来开启它。

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

2.1 什么是 GDT 表

GDT 全称是 Global Descriptor Table，中文名称叫“全局描述符表”，想要在“保护模式”下对内存进行寻址就先要有 GDT。GDT 表里的每一项叫做“段描述符”，用来记录每个内存分段的一些属性信息，每个“段描述符”占 8 字节。

在保护模式下，我们通过设置 GDT 将内存空间被分割为了一个又一个的段(这些段是可以重叠的)，这样我们就能实现不同的程序访问不同的内存空间。这和实模式下的寻址方式是不同的，在实模式下我们只能使用 $\text{address} = \text{segment} \ll 4 \mid \text{offset}$ 的方式进行寻址(虽然也是 $\text{segment} + \text{offset}$ 的，但在实模式下我们并不会真正的进行分段)。在这种情况下，任何程序都能访问整个 1MB 的空间。而在保护模式下，通过分段的方式，程序并不能访问整个内存空间

2.2 初始化 GDT 表

一个简单的 GDT 表和其描述符已经静态储存在引导区中，载入即可

```
lgdt gdttdesc
```

3 如何进入保护模式

通过将 cr0 寄存器 PE 位置 1 便开启了保护模式

```
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0
```

通过长跳转更新 cs 的基地址

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

```
protcseg:
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
movl $start, %esp
转到保护模式完成，进入 boot 主方法
```

练习 4

ELF 文件是什么？

ELF (Executable and linking format) 文件格式是 Linux 系统下的一种常用目标文件 (object file) 格式，有三种主要类型：

- 用于执行的可执行文件 (executable file)，用于提供程序的进程映像，加载到内存执行。这也是本实验的 OS 文件类型。
- 用于连接的可重定位文件 (relocatable file)，可与其它目标文件一起创建可执行文件和共享目标文件。
- 共享目标文件 (shared object file)，连接器可将它与其它可重定位文件和共享目标文件连接成其它的目标文件，动态连接器又可将它与可执行文件和其它共享目标文件结合起来创建一个进程映像。

ELF 文件有两种视图 (View)，链接视图和执行视图，如下图：

Linking View	Execution View
ELF header	ELF header
Program header table <i>optional</i>	Program header table
Section 1	Segment 1
...	
Section n	Segment 2
...	
...	...
Section header table	Section header table <i>optional</i>

链接视图通过 Section Header Table 描述，执行视图通过 Program Header Table 描述。Section Header Table 描述了所有 Section 的信息，包括所在的文件偏移和大小等；Program Header Table 描述了所有 Segment 的信息，即 Text Segment, Data Segment 和 BSS Segment，每个 Segment 中包含了一个或多个 Section。

对于加载可执行文件，我们只需关注执行视图，即解析 ELF 文件，遍历 Program Header Table 中的每一项，把每个 Program Header 描述的 Segment 加载到对应的虚拟地址即可，然后从 ELF header 中取出 Entry 的地址，跳转过去就开始执行了。对于 ELF 格式的内核文件来说，这个工作就需要由 Bootloader 完成。Bootloader 支持 ELF 内核文件加载之后，用 C 语言编写的内核编译完成之后就不需要 objcopy 了。

代码分析

```
#include <defs.h>
#include <x86.h>
#include <elf.h>
/*
    bootasm.S 完成了 bootloader 的大部分功能
    包括打开 A20，初始化 GDT，进入保护模式，更新段寄存器的值，建立堆栈
*/

/* *****
 * 这是一个非常简单的引导加载程序，它的唯一工作就是引导
 * 来自第一个 IDE 硬盘的 ELF 内核映像
 *
```

```

* 磁盘布局
* 这个程序(bootasm)。S 和 bootmain.c) 是引导加载程序。
* 应该存储在磁盘的第一个扇区。
*
** 第二个扇区包含内核映像。
*
** 内核映像必须是 ELF 格式。
*
* 开机步骤
** 当 CPU 启动时，它将 BIOS 加载到内存中并执行它
*
** BIOS 初始化设备，设置中断例程，以及
* 读取启动设备（硬盘）的第一个扇区
* 进入内存并跳转到它。
*
** Assuming this boot loader is stored in the first sector of the
* hard - drive, this code takes over...
*
** 控制启动 bootasm.S -- 设置保护模式，
* 和一个堆栈，C 代码然后运行，然后调用 bootmain()
*
** bootmain() 在这个文件中接管，读取内核并跳转到它
**/
unsigned int    SECTSIZE =      512 ;
struct elfhdr * ELFHDR    =      ((struct elfhdr *)0x10000) ;      // scratch space

/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}

/* readsect - read a single sector at @secno into @dst */
//读取扇区的代码如下：
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1);                                // count = 1   outb(使用内联汇编实现),
    设置读取扇区的数目为 1
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);

```

```

    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    //上面四条指令联合制定了扇区号在这4个字节联合构成的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
    outb(0x1F7, 0x20); // 0x20 命令, 读取扇区
    // wait for disk to be ready
    waitdisk();
    //将扇区内容加载到内存中虚拟地址 dst
    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4); //也用内联汇编实现
}

// wait for disk to be ready
waitdisk();

// read a sector
insl(0x1F0, dst, SECTSIZE / 4); //读到 dst 位置, 幻数(直接使用的常数叫做
幻数)4 因为这里以 dw 为单位
}

```

//从 outb() 可以看出这里是用 LBA 模式的 PIO (Program IO) 方式来访问硬盘的 (即所有的 IO 操作是通过 CPU 访问硬盘的 IO 地址寄存器完成), 该函数一次只读一个扇区

/*
 读取硬盘扇区的步骤:

1. 等待硬盘空闲。waitdisk 的函数实现只有一行: while ((inb(0x1F7) & 0xC0) != 0x40), 意思是不断查询读 0x1F7 寄存器的最高两位, 直到最高位为 0、次高位为 1 (这个状态应该意味着磁盘空闲) 才返回。
2. 硬盘空闲后, 发出读取扇区的命令。对应的命令字为 0x20, 放在 0x1F7 寄存器中; 读取的扇区数为 1, 放在 0x1F2 寄存器中; 读取的扇区起始编号共 28 位, 分成 4 部分依次放在 0x1F3~0x1F6 寄存器中。
3. 发出命令后, 再次等待硬盘空闲。
4. 硬盘再次空闲后, 开始从 0x1F0 寄存器中读数据。注意 insl 的作用是 "That function will read cnt dwords from the input port specified by port into the supplied output array addr.", 是以 dword 即 4 字节为单位的, 因此这里 SECTSIZE 需要除以 4。

```
    */
/* *
 * readseg - read @count bytes at @offset from kernel into virtual address @va,
 * might copy more than asked.
 * */
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // round down to sector boundary
    va -= offset % SECTSIZE;

    // translate from bytes to sectors; kernel starts at sector 1
    uint32_t secno = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for (; va < end_va; va += SECTSIZE, secno++) {
        readsect((void *)va, secno);
    }
}

/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // read the 1st page off disk
    //读取 ELF 的头部
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    //判断是否是合法的 ELF 文件
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags)
    // ELF 头部有描述 ELF 文件应加载到内存什么位置的描述表,
    // 先将描述表的头地址存在 ph
```

```

    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    //按照描述表将 ELF 文件中数据载入缓存
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
    }

    // call the entry point from the ELF header
    // note: does not return
    // ELF 文件 0x1000 位置后面的 0xd1ec 比特被载入内存 0x00100000
    // ELF 文件 0xf000 位置后面的 0xd20 比特被载入内存 0x0010e000
    // 根据 ELF 头部储存的入口信息，找到内核的入口
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();
    //跳到内核程序入口地址，将 cpu 控制权交给 ucore 内核代码
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}
/*
* bootloader 是如何加载 ELF 格式的 OS?
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；
   首先看 readsect 函数， readsect 从设备的第 secno 扇区读取数据到 dst 位置
   readseg 简单包装了 readsect，可以从设备读取任意长度的内容。
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。
*/

```

回答问题：

- **bootloader** 如何读取硬盘扇区的？

读硬盘扇区的代码如下

```

// wait for disk to be ready
waitdisk();
//读取扇区内容
outb(0x1F2, 1); // count = 1 outb(使用内联
汇编实现), 设置读取扇区的数目为 1
outb(0x1F3, secno & 0xFF);
outb(0x1F4, (secno >> 8) & 0xFF);
outb(0x1F5, (secno >> 16) & 0xFF);
outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);

```

```

        outb(0x1F7, 0x20);                // cmd 0x20 - read sectors
// 上面四条指令联合制定了扇区号    // 在这 4 个字节联合构成的 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
    // wait for disk to be ready
    waitdisk();
    //将扇区内容加载到内存中虚拟地址 dst
    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);        //也用内联汇编实现
}

```

读取硬盘扇区的步骤:

1. 等待硬盘空闲。waitdisk 的函数实现只有一行: while ((inb(0x1F7) & 0xC0) != 0x40), 意思是不断查询读 0x1F7 寄存器的最高两位, 直到最高位为 0、次高位为 1 (这个状态应该意味着磁盘空闲) 才返回。
2. 硬盘空闲后, 发出读取扇区的命令。对应的命令字为 0x20, 放在 0x1F7 寄存器中; 读取的扇区数为 1, 放在 0x1F2 寄存器中; 读取的扇区起始编号共 28 位, 分成 4 部分依次放在 0x1F3~0x1F6 寄存器中。
3. 发出命令后, 再次等待硬盘空闲。
4. 硬盘再次空闲后, 开始从 0x1F0 寄存器中读数据。注意 insl 的作用是 “That function will read cnt dwords from the input port specified by port into the supplied output array addr.”, 是以 dword 即 4 字节为单位的, 因此这里 SECTIZE 需要除以 4。

- bootloader 是如何加载 ELF 格式的 OS?

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

找到 print_stackframe 函数, 发现函数里面的注释已经提供了十分详细的步骤, 基本上按照提示来做就行了。

- 首先定义两个局部变量 ebp、esp 分别存放 ebp、esp 寄存器的值。这里将 ebp 定义为指针, 是为了方便后面取 ebp 寄存器的值。
- 调用 read_ebp 函数来获取执行 print_stackframe 函数时 ebp 寄存器的值, 这里 read_ebp 必须定义为 inline 函数, 否则获取的是执行 read_ebp 函数时的 ebp 寄存器的值。

- 调用 `read_eip` 函数来获取当前指令的位置,也就是此时 `eip` 寄存器的值。这里 `read_eip` 必须定义为常规函数而不是 `inline` 函数,因为这样的话在调用 `read_eip` 时会把当前指令的下一条指令的地址(也就是 `eip` 寄存器的值)压栈,那么在进入 `read_eip` 函数内部后便可以从栈中获取到调用前 `eip` 寄存器的值。
- 由于变量 `eip` 存放的是下一条指令的地址,因此将变量 `eip` 的值减去 1,得到的指令地址就属于当前指令的范围了。由于只要输入的地址属于当前指令的起始和结束位置之间, `print_debuginfo` 都能搜索到当前指令,因此这里减去 1 即可。
- 以后变量 `eip` 的值就不能再调用 `read_eip` 来获取了(每次调用获取的值都是相同的),而应该从 `ebp` 寄存器指向栈中的位置再往上一个单位中获取。
- 由于 `ebp` 寄存器指向栈中的位置存放的是调用者的 `ebp` 寄存器的值,据此可以继续顺藤摸瓜,不断回溯,直到 `ebp` 寄存器的值变为 0

代码如下:

```
void print_stackframe(void) {
    uint32_t *ebp = 0;
    uint32_t esp = 0;

    ebp = (uint32_t *)read_ebp();
    esp = read_eip();
    while (ebp)
    {
        cprintf("ebp:0x%08x eip:0x%08x args:", (uint32_t)ebp, esp);
        cprintf("0x%08x 0x%08x 0x%08x 0x%08x\n", ebp[2], ebp[3], ebp[4],
            ebp[5]);
        print_debuginfo(esp - 1);

        esp = ebp[1];
        ebp = (uint32_t *)*ebp;
    }
    /* LAB1 YOUR CODE : STEP 1 */
    /* (1) call read_ebp() to get the value of ebp. the type is
    (uint32_t);
    * (2) call read_eip() to get the value of eip. the type is
    (uint32_t);
    * (3) from 0 .. STACKFRAME_DEPTH
    *     (3.1) printf value of ebp, eip
    *     (3.2) (uint32_t)calling arguments [0..4] = the contents in
    address (uint32_t)ebp +2 [0..4]
    *     (3.3) cprintf("\n");
    *     (3.4) call print_debuginfo(eip-1) to print the C calling
    function name and line number, etc.
    *     (3.5) popup a calling stackframe
```

```

        unsigned gd_args : 5;           // # args, 0 for interrupt/trap
gates
        unsigned gd_rsv1 : 3;           // reserved(should be zero I
guess)
        unsigned gd_type : 4;           // type(STS_{TG,IG32,TG32})
        unsigned gd_s : 1;              // must be 0 (system)
        unsigned gd_dpl : 2;            // descriptor(meaning new)
privilege level
        unsigned gd_p : 1;              // Present
        unsigned gd_off_31_16 : 16;     // high bits of offset in
segment
};

```

中断向量表一个表项占八个字节，其中 `gd_off_15_0 : 16` 和 `gd_off_31_16 : 16` 表示低偏移量和高偏移量再加上 `gd_ss : 16`; 为段选择子到 GDT 中找到段描述符可得基址，程序入口地址为基址+偏移量

问题二：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`。在 `idt_init` 函数中，依次对所有中断入口进行初始化。使用 `mmu.h` 中的 `SETGATE` 宏，填充 `idt` 数组内容。每个中断的入口由 `tools/vectors.c` 生成，使用 `trap.c` 中声明的 `vectors` 数组即可。

`idt_init` 函数代码如下：

```

.globl __alltraps
.globl vector0
vector0:
    pushl $0
    pushl $0
    jmp __alltraps
.globl vector1
vector1:
    pushl $0
    pushl $1
    jmp __alltraps
.globl vector2

```

```
vector2:
    pushl $0
    pushl $2
    jmp __alltraps
.globl vector3
vector3:
    pushl $0
    pushl $3
    jmp __alltraps
.globl vector4
vector4:
    pushl $0
    pushl $4
    jmp __alltraps
.globl vector5
vector5:
    pushl $0
    pushl $5
    jmp __alltraps
.globl vector6
vector6:
    pushl $0
    pushl $6
    jmp __alltraps
.globl vector7
vector7:
    pushl $0
    pushl $7
    jmp __alltraps
```

问题三：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”。

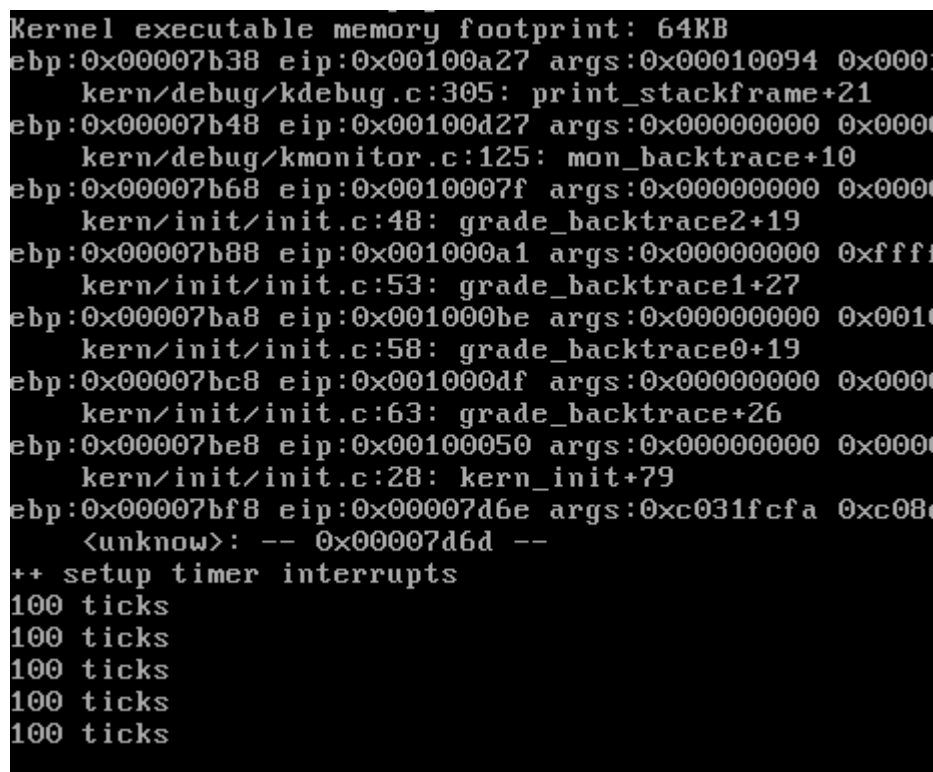
trap 函数的完整代码如下：

```
static void
trap_dispatch(struct trapframe *tf) {
    char c;

    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        ticks++;                //使用一个全局变量来记录时钟
```

```
        if (ticks == TICK_NUM) { //TICK_NUM 就是固定的 100，每到 100 便
调用 print_ticks() 函数
            ticks -= TICK_NUM;
            print_ticks();
        }
        break;
```

运行结果如图：



```
Kernel executable memory footprint: 64KB
ebp:0x00007b38 eip:0x00100a27 args:0x00010094 0x0000
    kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b48 eip:0x00100d27 args:0x00000000 0x0000
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b68 eip:0x0010007f args:0x00000000 0x0000
    kern/init/init.c:48: grade_backtrace2+19
ebp:0x00007b88 eip:0x001000a1 args:0x00000000 0xffff
    kern/init/init.c:53: grade_backtrace1+27
ebp:0x00007ba8 eip:0x001000be args:0x00000000 0x0010
    kern/init/init.c:58: grade_backtrace0+19
ebp:0x00007bc8 eip:0x001000df args:0x00000000 0x0000
    kern/init/init.c:63: grade_backtrace+26
ebp:0x00007be8 eip:0x00100050 args:0x00000000 0x0000
    kern/init/init.c:28: kern_init+79
ebp:0x00007bf8 eip:0x00007d6e args:0xc031fcfa 0xc080
    <unknown>: -- 0x00007d6d --
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

五、 实验收获与体会

本次实验覆盖了 BIOS 启动过程、bootloader 启动过程、保护模式和分段机制、硬盘访问、操作系统启动过程、函数堆栈、中断和异常等知识点，通过完成本次实验，我对课内学习的操作系统的关键的知识点有了更加深入的理解，同时，我也认识到我对 OS 的理解并没有足够的透彻，特别是没有完全弄清楚在 OS 实际运行中各种硬件软件之间的联系和硬件结构的不熟悉，这使得我在完成本次实验时遇到了许多困难。在接下来的学习中，我需要更加深入地理解理论，才能顺利完成接下来的实验。