

## 练习 1

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)

生成 kernel:

```
$(call add_files_cc,$(call listf_cc,$(LIBDIR)),libs,)
```

编译 libs 目录下所有的.c/.s, 生成.o/.d

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

编译 kernel 目录下所有.c/.s, 生成.o/.d, 具体指定了编译选项, 存放在 kcfldag 中, 其为:

-fno-builtin -nostdinc 关闭内建库

-fno-PIC 不生成位置无关的 symbol

-Wall 开启警告

-ggdb -gstabs 开启 gdb

-m32 目标平台 32 位

-fno-stack-protector 不生成栈保护

-o 输出

```
$(kernel): tools/kernel.ld
```

连接 kernel

生成 bootblock:

```
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc)
```

编译/boot 目录下的文件,

-nostdinc 不搜索默认路径头文件

-Os 优化生成代码

```
$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
```

连接 bootblock

-N 设置为读写

-e 设置入口

-Ttext 0x7C00 设置起始地址

```
@$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
```

OBJDUMP 反汇编, 保存至 asm 文件

```
@$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
```

OBJCOPY, 输出至 out 文件

-S 移除符号和重定位信息

-O binary 指定输出格式为 binary

```
@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
```

使用 sign 程序生成 bootblock

```
$(call add_files_host,tools/sign.c,sign,sign
```

生成 sign.o, 用于生成 bootblock 依赖

---

```
$(call create_target_host,sign,sign)
```

生成 sign, 用于生成 bootblock 依赖

生成 ucore.img:

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

从 /dev/zero 中获取 10000 个 block, 其为空字符

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

从 bootblock 中获取数据, 输出至 ucore.img

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

从 kernel 中获取数据, 跳过第一个 block, 输出至 ucore.img

至此, ucore.img 生成完毕。

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?

大小为 512 字节, 且以 0x55AA 结束。

## 练习 2

1 从 CPU 加电后执行的第一条指令开始, 单步跟踪 BIOS 的执行:

修改 lab1/tools/gbdinit:

```
set architecture i8086
```

```
target remote :1234
```

在 lab1 执行

```
make debug
```

在 gdb 界面执行

```
si //单步跟踪
```

```
x /2i $pc //显示当前 eip 处的汇编
```

```
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) x /2i $pc
=> 0xe05b:      add    %al, (%bx, %si)
   0xe05d:      add    %al, (%bx, %si)
(gdb) █
```

2 在初始化位置 0x7c00 设置实地址断点, 测试断点正常:

修改 gbdinit:

```
set architecture i8086
```

```
target remote :1234
```

```
b *0x7c00
```

```
c
```

```
x/2i $pc
```

断点正常：

```
The target architecture is assumed to be i8086
remote Thread 1
line: ?? PC: 0xffff0
0x0000ffff0 in ?? ()
Breakpoint 1 at 0x7c00

c0
Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:      cli
      0x7c01:      cld
(gdb)
```

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

改写 makefile:

```
218 debug: $(UCOREIMG) ^
219 #      $(V)$(QEMU) -S -s -parallel stdio -hda $< -serial null &
220 #      $(V)sleep 2
221 #      $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
222      $(V)$(TERMINAL) "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -parallel stdio -hda $< -
      serial null"
223      $(V)sleep 2
224      $(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
```

make debug 得到 q.log:

```
2 IN:
3 0x00007c00:  cli
4 0x00007c01:  cld
5 0x00007c02:  xor      %ax,%ax
6 0x00007c04:  mov      %ax,%ds
7 0x00007c06:  mov      %ax,%es
8 0x00007c08:  mov      %ax,%ss|
```

对比 bootasm.s, bootblock.asm 可以发现, bootasm.s 与 bootblock.s 一致, 与 q.log 中代码也一致。

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。

设置断点 0x7c4a, 在 gdb 中调试:

```
(gdb) c
Continuing.

Breakpoint 1, 0x00007c4a in ?? ()
(gdb) x/2i $pc
=> 0x7c4a:      call    0x7ccf
      0x7c4d:      add     %al,(%bx,%si)
(gdb)
```

---

## 练习 3 分析 bootloader 进入保护模式的过程

查看 bootasm.S，逐段分析：

在 0x7c00 加载 bootloader，关闭中断，初始化寄存器

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader

.globl start

start:

.code16

cli

cld

# 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

使能 A20：等待 8042 输入缓冲区为空，往 0x64 写入 0xd1，修改 8042 的 P2 端口，等待输入缓冲区为空，往 0x60 写入 0xDF，将 A20 置位为 1，开启 A20，利用 32 位寻址能力。

# 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

testb \$0x2, %al

jnz seta20.1

movb \$0xd1, %al

outb %al, \$0x64

seta20.2:

inb \$0x64, %al

testb \$0x2, %al

jnz seta20.2

movb \$0xdf, %al

outb %al, \$0x60

使用 GDT 从实模式切换到保护模式：载入 GDT 表，加载 cr0 到 eax，将 eax 的第 0 位置 1，将 cr0 的第 0 位置 1，长跳转更新 cs 基址。至此切换至保护模式。

```

# 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

```

## 练习 4 分析 bootloader 加载 ELF 格式 OS 的过程

### 1 读取扇区(readsect)

```

#等待 IO 不忙（等待磁盘准备好）
#向 IO 0x1f2/0x1f3/0x1f4/0x1f5/0x1f6/0x1f7 中写入参数（发出读取扇区命令）
#等待 IO 不忙（等待磁盘准备好）
#使用 insl 命令，读取硬盘数据至内存

```

### 2 加载 ELF 格式的 OS

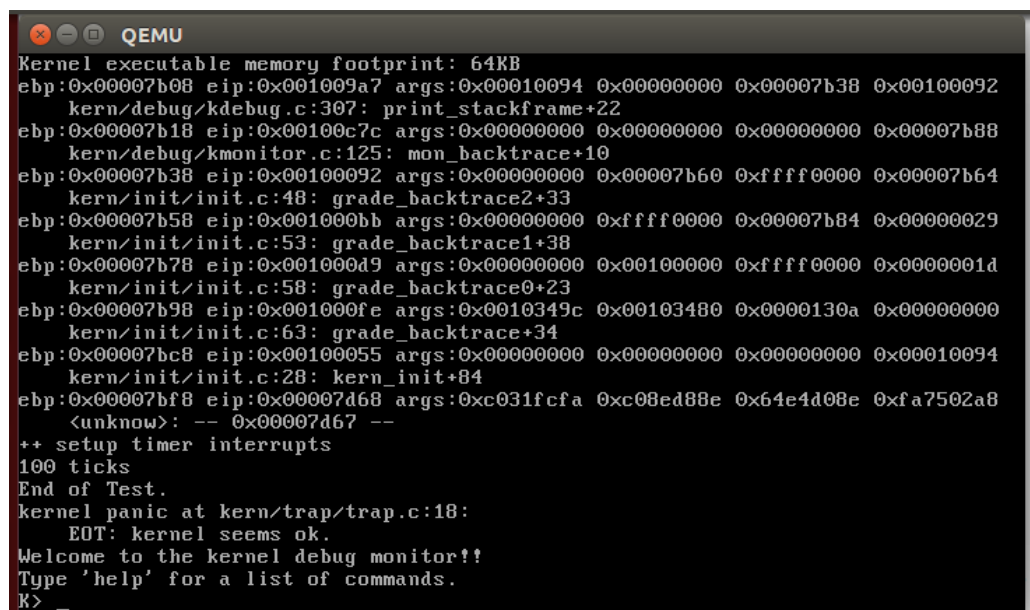
```

#读取 8 个扇区数据至 0x10000，转化成 elfhdr
#校验 e_magic
#根据 offset 将程序段数据读取至内存

```

## 练习 5 实现 print\_stackframe

代码另附，翻译 code 中的注释实现，make qemu 得到：



```

QEMU
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a7 args:0x00010094 0x00000000 0x00007b38 0x00100092
kern/debug/kdebug.c:307: print_stackframe+22
ebp:0x00007b18 eip:0x00100c7c args:0x00000000 0x00000000 0x00000000 0x00007b88
kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b38 eip:0x00100092 args:0x00000000 0x00007b60 0xffff0000 0x00007b64
kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b58 eip:0x001000bb args:0x00000000 0xffff0000 0x00007b84 0x00000029
kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b78 eip:0x001000d9 args:0x00000000 0x00100000 0xffff0000 0x0000001d
kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007b98 eip:0x001000fe args:0x0010349c 0x00103480 0x0000130a 0x00000000
kern/init/init.c:63: grade_backtrace+34
ebp:0x00007bc8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00010094
kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d67 --
++ setup timer interrupts
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>

```

简要过程:

```
#读取 ebp,eip
```

## #打印相关信息

### #查找调用者, 更新 ebp,eip

#ebp 非 0 且小于 STACKFRAME\_DEPTH 则循环

最后一行输出为第一个使用堆栈的函数(`bootmain`)。

## 练习 6 完善中断初始化和处理

1. 一个表项 8 字节，0-15 位为段偏移的 0-15 位，16-31 位为段选择子，48-63 位为段偏移的 16-31 位。这些位代表了中断处理代码的入口。
2. 完成代码后，运行程序如下图：

