Lab1 report

黄伟健 P14206019

[练习 1]

[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)

```
bin/ucore.img
| 生成 ucore.img 的相关代码为
| $(UCOREIMG): $(kernel) $(bootblock)
|       $(V)dd if=/dev/zero of=$@ count=10000
|       $(V)dd if=$(bootblock) of=$@ conv=notrunc
|       $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
|
| 为了生成 ucore.img，首先需要生成 bootblock、kernel
|
|>     bin/bootblock
|       | 生成 bootblock 的相关代码为
|       | $(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
|       |       @echo + ld $@
|       |       $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ \
|       |             -o $(call toobj,bootblock)
|       |       @$(OBJDUMP) -S $(call objfile,bootblock) > \
|       |             $(call asmfile,bootblock)
|       |       @$(OBJCOPY) -S -O binary $(call objfile,bootblock) \
|       |             $(call outfile,bootblock)
|       |       @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
|       |
|       | 为了生成 bootblock，首先需要生成 bootasm.o、bootmain.o、sign
|       |
|       |>     obj/boot/bootasm.o, obj/boot/bootmain.o
|       |       | 生成 bootasm.o,bootmain.o 的相关 makefile 代码为
|       |       | bootfiles = $(call listf_cc,boot)
|       |       | $(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),\
|       |       |       $(CFLAGS) -Os -nostdinc))
|       |       | 实际代码由宏批量生成
|       |       |
|       |       | 生成 bootasm.o 需要 bootasm.S
|       |       | 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
|       |       |       -m32   生成适用于 32 位环境的代码
|       |       |       -gstabs   生成 stabs 格式的调试信息
|       |       |       -nostdinc   不使用标准库
|       |       |       -fno-stack-protector   不生成用于检测缓冲区溢出的代码
```

```
|       |       |       -Os   为减小代码大小而进行优化
|       |       |       -I<dir>   添加搜索头文件的路径
|       |       |
|       |       | 生成 bootmain.o 需要 bootmain.c
|       |       | 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
|       |       | 新出现的关键参数有
|       |       |       -fno-builtin   除非用__builtin_前缀，
|       |       |                           否则不进行 builtin 函数的优化
|       |
|       |>      bin/sign
|       |       | 生成 sign 工具的 makefile 代码为
|       |       | $(call add_files_host,tools/sign.c,sign,sign)
|       |       | $(call create_target_host,sign,sign)
|       |       |
|       |       | 实际命令为
|       |       | 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
|       |
|       | 首先生成 bootblock.o
|       | 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
|       | 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
|
|>      bin/kernel
|       | 生成 kernel 的相关代码为
|       | $(kernel): tools/kernel.ld
|       | $(kernel): $(KOBJS)
|       |       @echo + ld $@
|       |       $(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)
|       |       @$(OBJDUMP) -S $@ > $(call asmfile,kernel)
|       |       @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; \
|       |               /^$$/d' > $(call symfile,kernel)
```

```
|       |
|       | 为了生成 kernel，首先需要 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
|       | kernel.ld 已存在
|       |
|       |>     obj/kern/*/*.o
|       |       | 生成这些.o 文件的相关 makefile 代码为
|       |       | $(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,\
|       |       |       $(KCFLAGS))
|       |       | 这些.o 生成方式和参数均类似，仅举 init.o 为例，其余不赘述
|       |>     obj/kern/init/init.o
|       |       | 编译需要 init.c
|       |       | 实际命令为
|       |       |       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
|       |
|       | 生成 kernel 时，makefile 的几条指令中有@前缀的都不必需
|       | 必需的命令只有
|       | 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
|       | 其中新出现的关键参数为
|       |       -T <scriptfile>    让连接器使用指定的脚本
|
| 生成一个有 10000 个块的文件，每个块默认 512 字节，用 0 填充
| dd if=/dev/zero of=bin/ucore.img count=10000
|
| 把 bootblock 中的内容写到第一个块
| dd if=bin/bootblock of=bin/ucore.img conv=notrunc
|
| 从第二个块开始写 kernel 中的内容
| dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
```

[练习 1.2] 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?

从 sign.c 的代码来看，一个磁盘主引导扇区只有 512 字节。且
第 510 个（倒数第二个）字节是 0x55，
第 511 个（倒数第一个）字节是 0xAA。

[练习 2]

[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。

通过改写 Makefile 文件 ()

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

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。
为防止 qemu 在 gdb 连接后立即开始执行，删除了 tools/gdbinit 中的"continue"行。

[练习 2.2] 在初始化位置 0x7c00 设置实地址断点,测试断点正常。

在 tools/gdbinit 结尾加上

```
set architecture i8086    //设置当前调试的 CPU 是 8086
b *0x7c00    //在 0x7c00 处设置断点。此地址是 bootloader 入口点地址，可看 boot/bootasm.S 的 start 地址处
c                //continue 简称，表示继续执行
x /2i $pc    //显示当前 eip 处的汇编指令
set architecture i386    //设置当前调试的 CPU 是 80386
```

运行"make debug"便可得到

```
Breakpoint 2, 0x00007c00 in ?? ()
=> 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
```

[练习 2.3] 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 tools/gdbinit 结尾加上

```
b *0x7c00
c
x /10i $pc
```

便可以在 q.log 中读到"call bootmain"前执行的命令

```
----------------
IN:
0x00007c00:  cli

----------------
IN:
```

```
0x00007c01:   cld
0x00007c02:   xor      %ax,%ax
0x00007c04:   mov      %ax,%ds
0x00007c06:   mov      %ax,%es
0x00007c08:   mov      %ax,%ss

----------------
IN:
0x00007c0a:   in       $0x64,%al

----------------
IN:
0x00007c0c:   test     $0x2,%al
0x00007c0e:   jne      0x7c0a

----------------
IN:
0x00007c10:   mov      $0xd1,%al
0x00007c12:   out      %al,$0x64
0x00007c14:   in       $0x64,%al
0x00007c16:   test     $0x2,%al
0x00007c18:   jne      0x7c14

----------------
IN:
0x00007c1a:   mov      $0xdf,%al
0x00007c1c:   out      %al,$0x60
0x00007c1e:   lgdtw    0x7c6c
0x00007c23:   mov      %cr0,%eax
0x00007c26:   or       $0x1,%eax
0x00007c2a:   mov      %eax,%cr0

----------------
IN:
0x00007c2d:   ljmp     $0x8,$0x7c32

----------------
IN:
0x00007c32:   mov      $0x10,%ax
0x00007c36:   mov      %eax,%ds

----------------
IN:
0x00007c38:   mov      %eax,%es

----------------
IN:
0x00007c3a:   mov      %eax,%fs
0x00007c3c:   mov      %eax,%gs
0x00007c3e:   mov      %eax,%ss

----------------
IN:
```

```
0x00007c40:    mov       $0x0,%ebp

----------------
IN:
0x00007c45:    mov       $0x7c00,%esp
0x00007c4a:    call      0x7d0d

----------------
IN:
0x00007d0d:    push      %ebp
```

其与 bootasm.S 和 bootblock.asm 中的代码相同。

[练习 3] 分析 bootloader 进入保护模式的过程。

从%cs=0 $pc=0x7c00，进入后

首先清理环境：包括将 flag 置 0 和将段寄存器置 0

```
.code16
    cli
    cld
    xorw %ax, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %ss
```

开启 A20：通过将键盘控制器上的 A20 线置于高电位，全部 32 条地址线可用，可以访问 4G 的内存空间。

```
seta20.1:                   # 等待 8042 键盘控制器不忙
    inb $0x64, %al          #
    testb $0x2, %al         #
    jnz seta20.1            #

    movb $0xd1, %al         # 发送写 8042 输出端口的指令
    outb %al, $0x64         #

seta20.1:                   # 等待 8042 键盘控制器不忙
    inb $0x64, %al          #
    testb $0x2, %al         #
    jnz seta20.1            #

    movb $0xdf, %al         # 打开 A20
    outb %al, $0x60         #
```

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 gdtdesc
```

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

```
        call bootmain
```

[练习 4] ：分析 bootloader 加载 ELF 格式的 OS 的过程。

首先看 readsect 函数，

readsect 从设备的第 secno 扇区读取数据到 dst 位置

```
    static void
    readsect(void *dst, uint32_t secno) {
        waitdisk();

        outb(0x1F2, 1);                         // 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 个字节线联合构成的 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

        waitdisk();

        insl(0x1F0, dst, SECTSIZE / 4);         // 读取到 dst 位置，
                                                    // 幻数 4 因为这里以 DW 为单位
    }
```

readseg 简单包装了 readsect，可以从设备读取任意长度的内容。

```
    static void
    readseg(uintptr_t va, uint32_t count, uint32_t offset) {
        uintptr_t end_va = va + count;

        va -= offset % SECTSIZE;
```

```
        uint32_t secno = (offset / SECTSIZE) + 1;
        //  加 1 因为 0 扇区被引导占用
        // ELF 文件从 1 扇区开始

        for (; va < end_va; va += SECTSIZE, secno ++) {
            readsect((void *)va, secno);
        }
    }
```

在 bootmain 函数中，

```
    void
    bootmain(void) {
        //  首先读取 ELF 的头部
        readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

        //  通过储存在头部的幻数判断是否是合法的 ELF 文件
        if (ELFHDR->e_magic != ELF_MAGIC) {
            goto bad;
        }

        struct proghdr *ph, *eph;

        // 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 文件中数据载入内存
        for (; ph < eph; ph ++) {
            readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
        }
        // 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 & 0xFFFFFF))();

    bad:
        outw(0x8A00, 0x8A00);
        outw(0x8A00, 0x8E00);
        while (1);
    }
```

[练习 5] 实现函数调用堆栈跟踪函数

ss:ebp 指向的堆栈位置储存着 caller 的 ebp，以此为线索可以得到所有使用堆栈的函数 ebp。
ss:ebp+4 指向 caller 调用时的 eip，ss:ebp+8 等是（可能的）参数。

输出中，堆栈最深一层为

```
ebp:0x00007bf8 eip:0x00007d68 \
    args:0x00000000 0x00000000 0x00000000 0x00007c4f
   <unknow>: -- 0x00007d67 --
```

其对应的是第一个使用堆栈的函数，bootmain.c 中的 bootmain。
bootloader 设置的堆栈从 0x7c00 开始，使用"call bootmain"转入 bootmain 函数。
call 指令压栈，所以 bootmain 中 ebp 为 0x7bf8。

[练习 6] 完善中断初始化和处理

[练习 6.1] 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？

中断向量表一个表项占用 8 字节，其中 2-3 字节是段选择子，0-1 字节和 6-7 字节拼成位移，两者联合便是中断处理程序的入口地址。

[练习 6.2] 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。
见代码

[练习 6.3] 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
见代码

[练习 7] 增加 syscall 功能，即增加一用户态函数（可执行一特定系统调用：获得时钟计数值），当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务

在 idt_init 中，将用户态调用 SWITCH_TOK 中断的权限打开。

```
SETGATE(idt[T_SWITCH_TOK], 1, KERNEL_CS, __vectors[T_SWITCH_TOK], 3);
```

在 trap_dispatch 中，将 iret 时会从堆栈弹出的段寄存器进行修改

对 TO User

```
tf->tf_cs = USER_CS;
tf->tf_ds = USER_DS;
tf->tf_es = USER_DS;
tf->tf_ss = USER_DS;
```

对 TO Kernel

```
tf->tf_cs = KERNEL_CS;
tf->tf_ds = KERNEL_DS;
tf->tf_es = KERNEL_DS;
```

在 lab1_switch_to_user 中，调用 T_SWITCH_TOU 中断。
注意从中断返回时，会多 pop 两位，并用这两位的值更新 ss,sp，损坏堆栈。
所以要先把栈压两位，并在从中断返回后修复 esp。

```
asm volatile (
    "sub $0x8, %%esp \n"
    "int %0 \n"
```

```
            "movl %%ebp, %%esp"
            :
            : "i"(T_SWITCH_TOU)
        );
```

在 lab1_switch_to_kernel 中，调用 T_SWITCH_TOK 中断。
注意从中断返回时，esp 仍在 TSS 指示的堆栈中。所以要在从中断返回后修复 esp。

```
        asm volatile (
            "int %0 \n"
            "movl %%ebp, %%esp \n"
            :
            : "i"(T_SWITCH_TOK)
        );
```

但这样不能正常输出文本。根据提示，在 trap_dispatch 中转 User 态时，将调用 io 所需权限降低。

```
        tf->tf_eflags |= 0x3000;
```