OS实验报告

Lab1：系统软件启动过程

PB15020603 蔡心宇

Content

环境搭建

因为之前一直在Reinforcement Leaning方面的内容，所以我的电脑很早之前就装了 `ubuntu 16.04` ，而且网上安装 `linux` & `windows` 双系统的教程多如牛毛而且质量大都还不错，在此不再赘述 `ubuntu` 的安装，关于环境用到的软件 `gcc`、`qemu` 等的安装实验指导书也有比较详细的介绍，也不再多说。

在这里主要想推荐一下一款开发工具 `Visual Studio Code` 简称 `VScode` 。

只是一款有Microsoft开发的免费跨平台轻量级文本编辑器，支持各种语言，插件丰富。有了这个就有了`代码编辑器`，`Meld`，`understand`，`Markdown`，`git`，`terminal` 等一系列必要的软件。

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,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

实验内容与结果

练习1

1.1 操作系统镜像文件 `ucore.img` 是如何一步一步生成的

执行 `make V=`

```
vegelofe@vegelofe-Surface-Book:~/OS/ucore_os_lab/labcodes/lab1$ make V=
+ cc kern/init/init.c
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
kern/init/init.c:95:1: warning: ‘lab1_switch_test’ defined but not used [-
Wunused-function]
 lab1_switch_test(void) {
 ^
+ cc kern/libs/stdio.c
gcc -Ikern/libs/ -fno-builtin -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
gcc -Ikern/libs/ -fno-builtin -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
gcc -Ikern/debug/ -fno-builtin -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’ [-Wimplicit-function-declaration]
```

```
     print_stackframe();
     ^
+ cc kern/debug/kdebug.c
gcc -Ikern/debug/ -fno-builtin -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]
 read_eip(void) {
 ^
+ cc kern/debug/kmonitor.c
gcc -Ikern/debug/ -fno-builtin -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
gcc -Ikern/driver/ -fno-builtin -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
gcc -Ikern/driver/ -fno-builtin -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
gcc -Ikern/driver/ -fno-builtin -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
gcc -Ikern/driver/ -fno-builtin -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
gcc -Ikern/trap/ -fno-builtin -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:14:13: warning: ‘print_ticks’ defined but not used [-
Wunused-function]
 static void print_ticks() {
             ^
kern/trap/trap.c:30:26: warning: ‘idt_pd’ defined but not used [-Wunused-
variable]
 static struct pseudodesc idt_pd = {
                          ^
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -fno-builtin -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
gcc -Ikern/trap/ -fno-builtin -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
gcc -Ikern/mm/ -fno-builtin -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
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/  -c libs/string.c -o obj/libs/string.o
+ cc libs/printfmt.c
gcc -Ilibs/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/  -c libs/printfmt.c -o obj/libs/printfmt.o
+ ld bin/kernel
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
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-
protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
+ 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
+ ld bin/bootblock
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: 488 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000
记录了10000+0 的读入
记录了10000+0 的写出
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0297356 s, 172 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 bytes copied, 0.000203984 s, 2.5 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
记录了146+1 的读入
记录了146+1 的写出
74828 bytes (75 kB, 73 KiB) copied, 0.000724565 s, 103 MB/s
```

从Terminal中输出的结果中看，执行了`gcc`、`ld` 和 `dd` 3种指令。

`gcc` 将各种 `.c` 文件和 `.S` 文件编译成为 `.o` 文件

`ld` 将 `.o` 文件链接成可执行程序

`dd` 利用 `kernel` 和 `bootblock` 生成最终的 `ucore.img` 。

但是值得注意的一点是，在 `/bin` 中还有一个 `sign` , 并没有发现有 `ld` 指令生成它， 仔细检查后发现 `sign` 是由 `gcc` 指令生成的

```
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
```

但是搜索 `sign` 并没有找到哪里用到了它，由于它出现在了链接 `bootblock` 的 `ld` 指令之前，于是打开 `Makefile` 查找 `sign` 果真在如下生成 `bootblock` 的指令中找到了 `sign` 。

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

可见先把 `bootblock.o` copy成了 `bootblock.out` 然后调用了sign来处理 `bootblock.out`

在Terminal的输出中，可以找到这两句

```
'obj/bootblock.out' size: 488 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
```

查看 `sign.c` 可以找到对应的这输出两句的代码，而且很容易可以发现 `sign` 的功能是将 `bootblock.out` 读入后扩充为512字节（补0），并且将最后两字节置为 `0x55aa` 写入 `bootblock` 中。

```
    printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st_size);

    ......//中间省略

    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]);
```

用 `xxd` 查看 `bootblock` 可见确实如此

```
vegelofe@vegelofe-Surface-Book:~/OS/ucore_os_lab/labcodes/lab1$ xxd
bin/bootblock
00000000: fafc 31c0 8ed8 8ec0 8ed0 e464 a802 75fa  ..1........d..u.
00000010: b0d1 e664 e464 a802 75fa b0df e660 0f01  ...d.d..u....`..
00000020: 166c 7c0f 20c0 6683 c801 0f22 c0ea 327c  .l|. .f...."..2|
00000030: 0800 66b8 1000 8ed8 8ec0 8ee0 8ee8 8ed0  ..f.............
00000040: bd00 0000 00bc 007c 0000 e8be 0000 00eb  .......|........
00000050: fe8d 7600 0000 0000 0000 0000 ffff 0000  ..v.............
00000060: 009a cf00 ffff 0000 0092 cf00 1700 547c  ..............T|
00000070: 0000 5589 e557 8d3c 1089 cac1 e909 5681  ..U..W.<......V.
00000080: e2ff 0100 008d 7101 5329 d053 897d f089  ......q.S).S.}..
00000090: c33b 5df0 7371 baf7 0100 00ec 83e0 c03c  .;].sq.........<
000000a0: 4075 f3ba f201 0000 b001 eeba f301 0000  @u..............
000000b0: 89f0 ee89 f0ba f401 0000 c1e8 08ee 89f0  ................
000000c0: baf5 0100 00c1 e810 ee89 f0ba f601 0000  ................
000000d0: c1e8 1883 e00f 83c8 e0ee b020 baf7 0100  ........... ....
000000e0: 00ee baf7 0100 00ec 83e0 c03c 4075 f389  ...........<@u..
000000f0: dfb9 8000 0000 baf0 0100 00fc f26d 81c3  .............m..
00000100: 0002 0000 46eb 8a58 5b5e 5f5d c355 31c9  ....F..X[^_].U1.
00000110: ba00 1000 00b8 0000 0100 89e5 5653 e84f  ............VS.O
00000120: ffff ff81 3d00 0001 007f 454c 4675 3fa1  ....=.....ELFu?.
00000130: 1c00 0100 0fb7 352c 0001 008d 9800 0001  ......5,........
00000140: 00c1 e605 01de 39f3 7318 8b43 088b 4b04  ......9.s..C..K.
00000150: 83c3 208b 53f4 25ff ffff 00e8 12ff ffff  .. .S.%.........
00000160: ebe4 a118 0001 0025 ffff ff00 ffd0 ba00  .......%........
00000170: 8aff ff89 d066 efb8 008e ffff 66ef ebfe  .....f......f...
00000180: 1400 0000 0000 0000 017a 5200 017c 0801  .........zR..|..
00000190: 1b0c 0404 8801 0000 2c00 0000 1c00 0000  ........,.......
000001a0: d2fe ffff 9b00 0000 0041 0e08 8502 420d  .........A....B.
000001b0: 0541 8703 4f86 0446 8305 027e c341 c641  .A..O..F...~.A.A
000001c0: c741 c50c 0404 0000 1c00 0000 4c00 0000  .A..........L...
000001d0: 3dff ffff 7300 0000 0041 0e08 8502 4e0d  =...s....A....N.
000001e0: 0542 8603 8304 0000 0000 0000 0000 0000  .B..............
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
```

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

根据上面的讨论，不难看出：符合规范的硬盘主引导扇区要有512字节而且最后两个字节为 `0x55aa`

练习2

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

在 `gdbinit` 中改为

```
set architecture i8086
target remote :1234
x /2i $pc
```

然后修改 `Makefile` 中的 `debug` 中的内容第一句

```
debug: $(UCOREIMG)
        # $(V)$(QEMU) -S -s -parallel stdio -hda $< -serial null &
    $(V)$(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"
```

执行 `make debug`

```
The target architecture is assumed to be i8086
0x0000fff0 in ?? ()
=> 0xfff0:      add    %al,(%bx,%si)
   0xfff2:      add    %al,(%bx,%si)
(gdb) si
0x0000e05b in ?? ()
(gdb) x /10i $pc
=> 0xe05b:      add    %al,(%bx,%si)
   0xe05d:      add    %al,(%bx,%si)
   0xe05f:      add    %al,(%bx,%si)
   0xe061:      add    %al,(%bx,%si)
   0xe063:      add    %al,(%bx,%si)
   0xe065:      add    %al,(%bx,%si)
   0xe067:      add    %al,(%bx,%si)
   0xe069:      add    %al,(%bx,%si)
   0xe06b:      add    %al,(%bx,%si)
   0xe06d:      add    %al,(%bx,%si)
(gdb) si
0x0000e062 in ?? ()
0x0000e066 in ?? ()
0x0000e068 in ?? ()
0x0000e06a in ?? ()
0x0000e070 in ?? ()
0x0000e076 in ?? ()
0x0000d165 in ?? ()
(gdb) x /10i $pc
=> 0xd165:      add    %al,(%bx,%si)
   0xd167:      add    %al,(%bx,%si)
   0xd169:      add    %al,(%bx,%si)
   0xd16b:      add    %al,(%bx,%si)
   0xd16d:      add    %al,(%bx,%si)
   0xd16f:      add    %al,(%bx,%si)
   0xd171:      add    %al,(%bx,%si)
   0xd173:      add    %al,(%bx,%si)
   0xd175:      add    %al,(%bx,%si)
   0xd177:      add    %al,(%bx,%si)
```

发现 gdb 中出现的并不是所期待的长跳指令，而后面给着的一系列指令都是没有任何意义的，但是查看 q.log 日志就会发现实际执行的指令并不是这些，第一条指令确实是位于 0xffffff0 的长跳指令。

2.2 在初始化位置 **0x7c00** 设置实地址断点,测试断点正常

将 gdbinit 修改为如下内容

```
set architecture i8086
target remote :1234
b *0x7c00
c
x /2i $pc
set architecture i386
```

执行 make debug 可以得到如下结果

```
The target architecture is assumed to be i8086
0x0000fff0 in ?? ()
Breakpoint 1 at 0x7c00

Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:      cli
   0x7c01:      cld
---Type <return> to continue, or q <return> to quit---
```

2.3 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

继续作如下操作

```
The target architecture is assumed to be i386
(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
```

与 bootasm.S 和 bootblock.asm 比较，发现他们是相同的

```
.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.
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
```

```
.code16                                             # Assemble for 16-bit
mode
    cli                                             # Disable interrupts
    7c00:       fa                      cli
    cld                                             # String operations
increment
    7c01:       fc                      cld

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    7c02:       31 c0                   xor    %eax,%eax
    movw %ax, %ds                                   # -> Data Segment
    7c04:       8e d8                   mov    %eax,%ds
    movw %ax, %es                                   # -> Extra Segment
    7c06:       8e c0                   mov    %eax,%es
    movw %ax, %ss                                   # -> Stack Segment
    7c08:       8e d0                   mov    %eax,%ss

00007c0a <seta20.1>:
    # 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).
    7c0a:       e4 64                   in     $0x64,%al
    testb $0x2, %al
    7c0c:       a8 02                   test   $0x2,%al
    jnz seta20.1
    7c0e:       75 fa                   jne    7c0a <seta20.1>

    movb $0xd1, %al                             # 0xd1 -> port 0x64
    7c10:       b0 d1                   mov    $0xd1,%al
    outb %al, $0x64                             # 0xd1 means: write
data to 8042's P2 port
    7c12:       e6 64                   out    %al,$0x64

00007c14 <seta20.2>:

seta20.2:
    inb $0x64, %al                              # Wait for not
busy(8042 input buffer empty).
    7c14:       e4 64                   in     $0x64,%al
    testb $0x2, %al
    7c16:       a8 02                   test   $0x2,%al
    jnz seta20.2
    7c18:       75 fa                   jne    7c14 <seta20.2>

    movb $0xdf, %al                             # 0xdf -> port 0x60
    7c1a:       b0 df                   mov    $0xdf,%al
    outb %al, $0x60                             # 0xdf = 11011111,
means set P2's A20 bit(the 1 bit) to 1
    7c1c:       e6 60                   out    %al,$0x60
```

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

将 `gdbinit` 中 `b *0x7c00` 改为 `b *0x100000` ，执行 `make debug` 得到如下结果

```
The target architecture is assumed to be i8086
0x0000fff0 in ?? ()
Breakpoint 1 at 0x100000

Breakpoint 1, 0x00100000 in ?? ()
=> 0x100000:    push   %bp
   0x100001:    mov    %sp,%bp
---Type <return> to continue, or q <return> to quit---
The target architecture is assumed to be i386
(gdb) x /10i $pc
=> 0x100000:    push   %ebp
   0x100001:    mov    %esp,%ebp
   0x100003:    sub    $0x18,%esp
   0x100006:    mov    $0x10fd20,%edx
   0x10000b:    mov    $0x10ea16,%eax
   0x100010:    sub    %eax,%edx
   0x100012:    mov    %edx,%eax
```

```
    0x100014:    sub    $0x4,%esp
    0x100017:    push   %eax
    0x100018:    push   $0x0
```

与 `kernel.asm` 比较，发现他们是一致的

```
int
kern_init(void) {
  100000:       55                      push   %ebp
  100001:       89 e5                   mov    %esp,%ebp
  100003:       83 ec 18                sub    $0x18,%esp
    extern char edata[], end[];
    memset(edata, 0, end - edata);
  100006:       ba 20 fd 10 00          mov    $0x10fd20,%edx
  10000b:       b8 16 ea 10 00          mov    $0x10ea16,%eax
  100010:       29 c2                   sub    %eax,%edx
  100012:       89 d0                   mov    %edx,%eax
  100014:       83 ec 04                sub    $0x4,%esp
  100017:       50                      push   %eax
  100018:       6a 00                   push   $0x0
```

练习3

3.1 使能 `A20` 地址线

根据附录中的，当 `80286` 出现时，为了完全向下兼容 `8086` 的实模式，需要屏蔽 `A20` 地址线，然而为了进入保护模式，必须使能 `A20` 地址线才能寻址超过1MB的空间。使能 `A20` 的操作如下

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;

对比 `bootasm.S` 中的代码，可以看出他们是一致的

```
    # Enable A20:
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
```

3.2 初始化 GDT 表

因为 保护模式 使用 分段储存管理机制 ，所以需要初始换 GDT 表，在使能 A20 的代码下面可以找相应的指令

```
lgdt gdtdesc
```

在最下面还能找到 GDT 表的相关初始化信息，包括空段、代码段和数据段

```
# 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

gdtdesc:
    .word 0x17                                      # sizeof(gdt) - 1
    .long gdt
```

3.3 使能和进入保护模式

接下来的操作就是使能保护模式，需要将 cr0 寄存器的 PE 位 置1，相关代码如下

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

此时虽然已经使能保护模式，但是 cs 中还不是 segment selector 执行

```
    ljmp $PROT_MODE_CSEG, $protcseg
```

可以将 PROT_MODE_CSEG 的值 0x8放入 cs 中，然后继续执行 protcseg 中初始化数据段和堆栈的指令，这就真正进入了保护模式

练习4

在初始化堆栈之后会有

```
    call bootmain
```

调用 `bootmain` 函数，它就是用来将硬盘中 `ELF` 格式的 `kernel` 加载到内存中

4.1 bootloader如何读取硬盘扇区的

1. 等待磁盘准备好
2. 发出读取扇区的命令
3. 等待磁盘准备好
4. 把磁盘扇区数据读到指定内存

在 `bootmain.c` 中可以找到 `readsect` 函数用于读取 `secno` 扇区内容到内存的 `dst` 处

```
/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1);                         // count = 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

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

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}
```

然后又用 `readseg` 函数将其封装，可以读取 `offset` 处的 `count` 个字节到内存 `va` 处

```
/* *
 * 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);
    }
}
```

4.2 bootloader是如何加载ELF格式的OS

封装好 `readseg` 函数后在 `bootmain` 中调用它，来加载 `ELF` 格式的 `OS`

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

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags)
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph ++) {
        readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
    }

    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

先加载第一个扇区，然后再将数据段全部加载进入内存，最后转到 `kernel` 的起始地址开始执行。

但是一些细节仍然需要思考，在一开始有 `ELFHDR` 的定义

```
#define ELFHDR          ((struct elfhdr *)0x10000)      // scratch space
```

可以看出它是一个指向 `elfhdr` 类型的结构体的指针，值为 `0x10000`

在 `bootblock.asm` 中查找下面这句代码的汇编指令

```
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

可以找到

```
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
    7d62:       a1 18 00 01 00          mov    0x10018,%eax
    7d67:       25 ff ff ff 00          and    $0xffffff,%eax
    7d6c:       ff d0                   call   *%eax
```

但我们并不知道 `0x10018` 和 `eax` 寄存器的值 通过查看 `q.log` 可以发现执行 `call` 指令后，跳到了 `0x100000` 处执行

```
IN:
0x00007d62:  mov    0x10018,%eax
0x00007d67:  and    $0xffffff,%eax
0x00007d6c:  call   *%eax

----------------
IN:
0x00100000:  push   %ebp
0x00100001:  mov    %esp,%ebp
0x00100003:  sub    $0x18,%esp
0x00100006:  mov    $0x10fd20,%edx
0x0010000b:  mov    $0x10ea16,%eax
0x00100010:  sub    %eax,%edx
0x00100012:  mov    %edx,%eax
0x00100014:  sub    $0x4,%esp
0x00100017:  push   %eax
0x00100018:  push   $0x0
0x0010001a:  push   $0x10ea16
0x0010001f:  call   0x102a26

----------------
```

而查看 `kernel.asm` 可以发现，这正是 `kernel` 的起始地址

```
int
kern_init(void) {
  100000:       55                      push   %ebp
  100001:       89 e5                   mov    %esp,%ebp
  100003:       83 ec 18                sub    $0x18,%esp
    extern char edata[], end[];
```

```
    memset(edata, 0, end - edata);
  100006:       ba 20 fd 10 00          mov    $0x10fd20,%edx
  10000b:       b8 16 ea 10 00          mov    $0x10ea16,%eax
```

至此，bootloader 的工作已经完成，接下来进入操作系统的内核。

练习5

5.1 实现函数调用堆栈跟踪函数

```
void
print_stackframe(void) {
     /* 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
      *           NOTICE: the calling funciton's return addr eip  = ss:
[ebp+4]
      *                   the calling funciton's ebp = ss:[ebp]
      */
    uint32_t ebp = read_ebp();
    uint32_t eip = read_eip();

    for(int i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i++)
    {
        cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        uint32_t *args = (uint32_t *)ebp + 2;
        for(int j = 0; j < 4; j++)
        {
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip -1);
        eip = *((uint32_t *)(ebp + 4));
        //eip = *((uint32_t *)ebp + 1);
        ebp = *((uint32_t *)ebp);
    }
}
```

注释已经写的很清楚，需要注意的就是，ebp 下面就是函数调用返回时需要执行的指令的地址，即 （ebp + 4） 处的值就是 eip。 而且 （ebp） 处存的值就是父函数 的 ebp 值。

```
vegelofe@vegelofe-Surface-Book:~/OS/ucore_os_lab/labcodes/lab1$ make qemu
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.
main-loop: WARNING: I/O thread spun for 1000 iterations
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x0010327c (phys)
  edata  0x0010ea16 (phys)
  end    0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b38 eip:0x00100a3c args:0x00010094 0x00010094 0x00007b68
0x0010007f
    kern/debug/kdebug.c:306: print_stackframe+21
ebp:0x00007b48 eip:0x00100d3c args:0x00000000 0x00000000 0x00000000
0x00007bb8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b68 eip:0x0010007f args:0x00000000 0x00007b90 0xffff0000
0x00007b94
    kern/init/init.c:48: grade_backtrace2+19
ebp:0x00007b88 eip:0x001000a1 args:0x00000000 0xffff0000 0x00007bb4
0x00000029
    kern/init/init.c:53: grade_backtrace1+27
ebp:0x00007ba8 eip:0x001000be args:0x00000000 0x00100000 0xffff0000
0x00100043
    kern/init/init.c:58: grade_backtrace0+19
ebp:0x00007bc8 eip:0x001000df args:0x00000000 0x00000000 0x00000000
0x00103280
    kern/init/init.c:63: grade_backtrace+26
ebp:0x00007be8 eip:0x00100050 args:0x00000000 0x00000000 0x00000000
0x00007c4f
    kern/init/init.c:28: kern_init+79
ebp:0x00007bf8 eip:0x00007d6e args:0xc031fcfa 0xc08ed88e 0x64e4d08e
0xfa7502a8
    <unknow>: -- 0x00007d6d --
++ setup timer interrupts
```

可以看到执行 `make qemu` 的输出确实符合函数调用树。

练习6

6.1 中断描述符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。

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**。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。

```
void
idt_init(void) {
    extern uintptr_t __vectors[];

    for (int i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++)
    {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    // load the IDT
    lidt(&idt_pd);
}
```

`IDT` 的所有内容在 `vectors.S` 中， `_vectors` 其实是在 `vectors.S` 中定义的，在这里声明了一下，然后使用宏 `SETGATE` 将所有装入 `idt`，最后调用 `lidt` 指令装载 `IDT`。

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”**。

这个任务比较简单 在 `vecters.S` 中可以发现每个中断服务程序后面都有 `jmp __alltraps` 指令，`__alltraps` 在 `trapentry.S` 中，可以发现它会调用 `trap`，所以每当终端发生，就会执行 `trap_dispatch`，在其中添加如下代码即可

```
    case IRQ_OFFSET + IRQ_TIMER:
        ticks++;
        if(ticks % TICK_NUM == 0)
            print_ticks();
        break;
```

扩展练习

7.1 扩展 **proj4**,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` 中添加

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

从而添加一个用户态的中断服务程序。

在 `lab1_switch_to_user` 中调用 `T_SWITCH_TOU` 中断

```
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
        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

```
static void
lab1_switch_to_kernel(void) {
    //LAB1 CHALLENGE 1 :  TODO
    asm volatile (
            "int %0 \n"
            "movl %%ebp, %%esp \n"
            :
            : "i"(T_SWITCH_TOK)
        );
}
```

最后对 trap_dispatch 中的 T_SWITCH_TOU 和 T_SWITCH_TOK 两个 case 进行修改

```
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
        if (tf->tf_cs != USER_CS) {
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;

            // set eflags, make sure ucore can use io under user mode.
            // if CPL > IOPL, then cpu will generate a general protection.
            switchk2u.tf_eflags |= FL_IOPL_MASK;

            // set temporary stack
            // then iret will jump to the right stack
            *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
        }
        break;
    case T_SWITCH_TOK:
        if (tf->tf_cs != KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
```

```
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct
trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
        }
        break;
```

7.2 用键盘实现用户模式内核模式切换。具体目标是:“键盘输入 **3** 时切换到用户模式,键盘输入 **0** 时切换到内核模式”。 基本思路是借鉴软中断**(syscall** 功能**)**的代码,并且把**trap.c** 中软中断处理的设置语句拿过来

在实现 `7.1` 的情况下，直接将用户态和内核态相互切换的放到 `case IRQ_OFFSET + IRQ_KBD` ，加上对键盘输入的判断即可

```
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        if(c == 51)
        {
            cprintf("kbd [%03d] %c\n", c, c);
            if (tf->tf_cs != USER_CS) {
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;

            // set eflags, make sure ucore can use io under user mode.
            // if CPL > IOPL, then cpu will generate a general protection.
            switchk2u.tf_eflags |= FL_IOPL_MASK;

            // set temporary stack
            // then iret will jump to the right stack
            *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
            }
            print_trapframe(tf);
        }
        else if (c == 48)
        {
            cprintf("kbd [%03d] %c\n", c, c);
            if (tf->tf_cs != KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct
trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
            }
            print_trapframe(tf);
        }
        else
```

```
            cprintf("kbd [%03d] %c\n", c, c);
        break;
```