ucore实验1

笔记本：	My Notebook		
创建时间：	2022/7/6 15:10	**更新时间：**	2022/7/9 0:17
作者：	vwc0bw40		
URL：	https://chyyuu.gitbooks.io/ucore_os_docs/content/lab1/lab1_2_1_1_ex1.html		

ucore实验1

作者：赵凌珂

练习1：理解通过make生成执行文件的过程。

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？

补充材料：

如何调试Makefile

当执行make时，一般只会显示输出，不会显示make到底执行了哪些命令。

如想了解make执行了哪些命令，可以执行：

```
$ make "V="
```

要获取更多有关make的信息，可上网查询，并请执行

```
$ man make
```

实验过程和结果：

1.操作系统镜像文件ucore.img的生成

执行make：

```
+ cc kern/init/init.c
+ cc kern/libs/readline.c
+ cc kern/libs/stdio.c
```

```
+ cc kern/debug/kdebug.c
+ cc kern/debug/kmonitor.c
+ cc kern/debug/panic.c
+ cc kern/driver/clock.c
+ cc kern/driver/console.c
+ cc kern/driver/intr.c
+ cc kern/driver/picirq.c
+ cc kern/trap/trap.c
+ cc kern/trap/trapentry.S
+ cc kern/trap/vectors.S
+ cc kern/mm/pmm.c
+ cc libs/printfmt.c
+ cc libs/string.c
+ ld bin/kernel
+ cc boot/bootasm.S
+ cc boot/bootmain.c
+ cc tools/sign.c
+ ld bin/bootblock
'obj/bootblock.out' size: 488 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.100628 s, 50.9 MB/s
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000931437 s, 550 kB/s
146+1 records in
146+1 records out
74923 bytes (75 kB) copied, 0.00384922 s, 19.5 MB/s
```

执行make V= 指令详细查看编译过程和指令：

```
//初始化
+ 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
//读行
+ 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
//标准IO
+ 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/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
+ 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/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

//时钟控制
+ 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/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/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/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
+ 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/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/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
//格式化输出
+ 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
//字符串相关
+ 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
//建立链接
+ ld bin/kernel
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

//构建bootblock
+ 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!

//构建ucore.img
//使用dd创建空文件
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.151391 s, 33.8 MB/s
//使用dd将bin/bootlock写入ucore.img
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
```

```
512 bytes (512 B) copied, 0.000233029 s, 2.2 MB/s
//使用dd将bin/kernel写入ucore.img的1个block之后，即bootlock之后
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
146+1 records in
146+1 records out
74923 bytes (75 kB) copied, 0.00185385 s, 40.4 MB/s
```

整个过程总结如下：

1. 构建kernel内核文件，将“.c”文件转换成“.o”文件；
2. 生成链接
3. 构建bootblock
4. 生成ucore.img

查看Makefile文件可知生成ucore.img文件的代码为：

```
# create ucore.img
UCOREIMG    := $(call totarget,ucore.img)


$(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
```

具体过程如下：
1.创建一个大小为 10000 字节的块
2.将 bootblock，kernel 拷贝过去。
3.通过 dd 命令将 bootblock 放到第一个 sector，将 kernel 放到第二个 sector 开始的区域。
bootblock 就是引导区，kernel 则是操作系统内核。

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？

执行less tools/sign.c：

```
1 #include <stdio.h>
      2 #include <errno.h>
      3 #include <string.h>
      4 #include <sys/stat.h>
      5
      6 int
      7 main(int argc, char *argv[]) {
      8     struct stat st;
      9     if (argc != 3) {
     10         fprintf(stderr, "Usage: <input filename> <output filename>\n");
     11         return -1;
     12     }

            //读取文件
     13     if (stat(argv[1], &st) != 0) {
     14         fprintf(stderr, "Error opening file '%s': %s\n", argv[1],
strerror(errno));
     15         return -1;
     16     }
```

```
        //输出文件大小
17      printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st_size);

        //检查文件大小，不得超过510
18      if (st.st_size > 510) {
19          fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
20          return -1;
21      }

        //定义缓冲区
22      char buf[512];
23      memset(buf, 0, sizeof(buf));
24      FILE *ifp = fopen(argv[1], "rb");
25      int size = fread(buf, 1, st.st_size, ifp);
26      if (size != st.st_size) {
27          fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
28          return -1;
29      }
30      fclose(ifp);

        //结束标志位
31      buf[510] = 0x55;
32      buf[511] = 0xAA;

        //写入文件
33      FILE *ofp = fopen(argv[2], "wb+");
34      size = fwrite(buf, 1, 512, ofp);

        //检查文件大小
35      if (size != 512) {
36          fprintf(stderr, "write '%s' error, size is %d.\n", argv[2],
size);
37          return -1;
38      }
39      fclose(ofp);//释放文件
40      printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
41      return 0;
42  }
43
```

由编码可以看出，硬盘主引导扇区的特征是：

1.磁盘主引导扇区只有 512 字节；
2.内容不超过 510 字节；
3.最后两个字节为 0x55和0xAA。

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。

> 提示：参考附录“启动后第一条执行的指令”，可了解更详细的解释，以及如何单步调试和查看BIOS代码。
>
> 提示：查看 labcodes_answer/lab1_result/tools/lab1init 文件，用如下命令试试如何调试bootloader第一条指令：
>
> ```
> $ cd labcodes_answer/lab1_result/
> $ make lab1-mon
> ```

补充材料： 我们主要通过硬件模拟器qemu来进行各种实验。在实验的过程中我们可能会遇上各种各样的问题，调试是必要的。qemu支持使用gdb进行的强大而方便的调试。所以用好qemu和gdb是完成各种实验的基本要素。

默认的gdb需要进行一些额外的配置才进行qemu的调试任务。qemu和gdb之间使用网络端口1234进行通讯。在打开qemu进行模拟之后，执行gdb并输入

```
target remote localhost:1234
```

即可连接qemu，此时qemu会进入停止状态，听从gdb的命令。

另外，我们可能需要qemu在一开始便进入等待模式，则我们不再使用make qemu开始系统的运行，而使用make debug来完成这项工作。这样qemu便不会在gdb尚未连接的时候擅自运行了。

gdb的地址断点

在gdb命令行中，使用b *[地址]便可以在指定内存地址设置断点，当qemu中的cpu执行到指定地址时，便会将控制权交给gdb。

关于代码的反汇编

有可能gdb无法正确获取当前qemu执行的汇编指令，通过如下配置可以在每次gdb命令行前强制反汇编当前的指令，在gdb命令行或配置文件中添加：

```
define hook-stop
x/i $pc
end
```

即可

gdb的单步命令

在gdb中，有next, nexti, step, stepi等指令来单步调试程序，他们功能各不相同，区别在于单步的“跨度”上。

```
next 单步到程序源代码的下一行，不进入函数。
nexti 单步一条机器指令，不进入函数。
step 单步到下一个不同的源代码行（包括进入函数）。
stepi 单步一条机器指令。
```

实验过程和结果：

1.单步跟踪BIOS的执行

1.执行less lab1init命令：

```
1 file bin/kernel
2 target remote :1234
3 set architecture i8086
4 b *0x7c00
5 continue
6 x /2i $pc
```

其中，b即break ,设置断点于0x7c00位置

2.执行make lab1-mon：

```
0x0000fff0 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
Breakpoint 1 at 0x7c00

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

可知，断点在0x7c00处，设置成功。

3.在下方的gdb中输入x/10i $pc查看之后的十条指令：

```
=> 0x7c00 <start>:      cli
   0x7c01 <start+1>:    cld
   0x7c02 <start+2>:    xor    %ax,%ax
   0x7c04 <start+4>:    mov    %ax,%ds
   0x7c06 <start+6>:    mov    %ax,%es
   0x7c08 <start+8>:    mov    %ax,%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
```

与bootasm.S中代码进行比较：

```
# 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.
seta20.1:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.1
```

可见代码一致。

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。

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：

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

实验过程和结果：

- 早期的8086 CPU提供了20根地址线,可寻址空间范围为1MB内存空间。8086的数据处理位宽位16位，无法直接寻址1MB内存空间，所以8086提供了段地址加偏移地址的地址转换机制。
- PC机的寻址结构是segment:offset，segment和offset都是16位的寄存器，换算成物理地址的计算方法是把segment左移4位，再加上offset，所以segment:offset所能表达的寻址空间最大应为0ffff0h + 0ffffh =

10ffefh，大约是1088KB。这意味着segment:offset的地址表示能力，超过了20位地址线的物理寻址能力。

- 当寻址到超过1MB的内存时，会发生“回卷”（不会发生异常）。
- 下一代的基于Intel 80286 CPU的PC AT计算机系统提供了24根地址线，这样CPU的寻址范围变为 2^24=16M,同时也提供了保护模式，可以访问到1MB以上的内存。
- 但是这也造成了向下不兼容。为了保持完全的向下兼容性，于是出现了A20 Gate。
- 把A20地址线控制和键盘控制器的一个输出进行AND操作，这样来控制A20地址线的打开（使能）和关闭（屏蔽\禁止）。一开始时A20地址线控制是被屏蔽的（总为0），直到系统软件通过一定的IO操作去打开它（参看bootasm.S）。很显然，在实模式下要访问高端内存区，这个开关必须打开，在保护模式下，由于使用32位地址线，如果A20恒等于0，那么系统只能访问奇数兆的内存，即只能访问0--1M、2-3M、4-5M.....，这样无法有效访问所有可用内存。所以在保护模式下，这个开关也必须打开。

查看bootasm.S的代码：

```
#include <asm.h>
# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG,        0x8                     # kernel code segment
selector
.set PROT_MODE_DSEG,        0x10                    # kernel data segment
selector
.set CR0_PE_ON,             0x1                     # protected mode enable flag

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

//实模式启动执行,告诉编译器使用16位模式编译,然后关中断,最后将段寄存器清零
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.

//在保护模式下,将A20的置于高电位,使得A20的32条地址线全部可用
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.

//加载GDT表
    lgdt gdtdesc

//进入保护模式
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax
    movl %eax, %cr0
```

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？

提示：可阅读“硬盘访问概述”，“ELF执行文件格式概述”这两小节。

实验过程和结果：

查看bootmain.c代码：

```
#include <defs.h>
#include <x86.h>
#include <elf.h>

//bootloader读取磁盘扇区的代码
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready//等待磁盘扇区就位
    waitdisk();

//发出读取磁盘的命令,下面指令联合制定了扇区号
    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);
}

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


//.判断ELF格式的文件并放入内存
void
bootmain(void) {

//判断是否为ELF格式的文件
    // 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;
    }

//判断加载的内存位置
    struct proghdr *ph, *eph;

//加入内存
    // 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);
    }

//跳入内核的入口并将控制权全部交给内核
    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}
```

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。在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 0x0
kern/debug/kdebug.c:305: print_stackframe+22
ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000 0x00000000 0x0
kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80 0xffff0000 0x0
kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000 0x00007ba4 0x0
kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000 0xffff0000 0x0
kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520 0x00001308 0x0
kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x0
kern/init/init.c:28: kern_init+88
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xf
<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。

实验过程和结果：

查看kdebug.c的代码注释：

```
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
(unit32_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(), eip = read_eip();


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

执行make qemu命令：

```
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00010094 0x00000000 0x00007b38 0x00100092
    kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b18 eip:0x00100c95 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:0x001032fc 0x001032e0 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
    <unknow>: -- 0x00007d67 --
```

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”。

> 【注意】除了系统调用中断(T_SYSCALL)使用陷阱门描述符且权限为用户态权限以外，其它中断均使用特权级(DPL)为 0 的中断门描述符，权限为内核态权限；而ucore的应用程序处于特权级 3 ，需要采用 ` int 0x80` 指令操作（这种方式称为软中断，软件中断，Tra中断，在lab5会碰到）来发出系统调用请求，并要能实现从特权级 3 到特权级 0 的转换，所以系统调用中断(T_SYSCALL)所对应的中断门描述符中的特权级（DPL）需要设置为 3 。

要求完成问题2和问题3 提出的相关函数实现，提交改进后的源代码包（可以编译执行），并在实验报告中简要说明实现过程，并写出对问题1的回答。完成这问题2和3要求的部分代码后，运行整个系统，可以看到大约每1秒会输出一次”100 ticks”，而按下的键也会在屏幕上显示。

提示：可阅读小节“中断与异常”。

实验过程和结果：

1.中断描述符表，打开mmu.h代码：

```
/* Gate descriptors for interrupts and traps */
struct gatedesc {
    unsigned gd_off_15_0 : 16;        // low 16 bits of offset in segment
    unsigned gd_ss : 16;            // segment selector
    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
};
```

2.完善kern/trap/trap.c中对中断向量表进行初始化的函数idt_init

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    // set for switch from user to kernel
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    // load the IDT
```

```
    lidt(&idt_pd);
}
```

3.完善trap.c中的中断处理函数trap

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

    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a
global variable (increase it), such as ticks in kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a funciton,
such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        ticks ++;
        if (ticks % TICK_NUM == 0) {
            print_ticks();
        }
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    //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;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    default:
        // in kernel, it must be a mistake
        if ((tf->tf_cs & 3) == 0) {
            print_trapframe(tf);
```

```
            panic("unexpected trap in kernel.\n");
        }
    }
}


/* *
* trap - handles or dispatches an exception/interrupt. if and when trap()
returns,
* the code in kern/trap/trapentry.S restores the old CPU state saved in the
* trapframe and then uses the iret instruction to return from the exception.
* */
void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}
```

执行make qemu：

```
Kernel executable memory footprint: 64KB
ebp:0x00007b08 eip:0x001009a6 args:0x00010094 0x00000000 0x00007b38 0x00100092
    kern/debug/kdebug.c:305: print_stackframe+21
ebp:0x00007b18 eip:0x00100c95 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:0x0010359c 0x00103580 0x0000136a 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
    <unknow>: -- 0x00007d67 --
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```