

练习 1：理解通过 make 生成执行文件的过程。

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）

执行命令 `make V=`

实际上是设置一个标记，使得 Make 它的执行过程能够展现出来

那可以看到它调了 GCC，把一些 C 的源代码编译成了 .o 结尾的文件。

```
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -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/ -march=i686 -fno-builtin -fno-PIC -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/ -march=i686 -fno-builtin -fno-PIC -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/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/string.c -o obj/libs/string.o
+ cc libs/printfmt.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -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/ -march=i686 -fno-builtin -fno-PIC -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/ -march=i686 -fno-builtin -fno-PIC -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
```

然后通过 `ld`，`ld` 会把这些目标文件转换成一个执行程序比如说在这里面会转换成这个 `bootblock.out`（一个 Bootloader 执行程序）

```
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: 484 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
```

最后用 `dd` 可以把这个 Bootloader 放到一个虚拟的硬盘里面去，生成一个虚拟硬盘叫 `ucore.img` count。

```
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0179711 s, 285 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
```

硬件模拟器呢会基于这个虚拟硬盘中的数据来执行相应的代码

这里面其实生成了两个软件

第一个是 Bootloader 第二叫 kernel, kernel 实际上是 uCore 的组成部分

```
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0179711 s, 285 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000137972 s, 3.7 MB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
146+1 records in
146+1 records out
74828 bytes (75 kB, 73 KiB) copied, 0.000348428 s, 215 MB/s
```

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?

硬盘主引导扇区的规范写在了 sign.c 文件中

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>

int
main(int argc, char *argv[]) {
    struct stat st;
    if (argc != 3) {
        fprintf(stderr, "Usage: <input filename> <output filename>\n");
        return -1;
    }
    if (stat(argv[1], &st) != 0) {
        fprintf(stderr, "Error opening file '%s': %s\n", argv[1], strerror(errno));
        return -1;
    }
    printf("%s' size: %lld bytes\n", argv[1], (long long)st.st_size);
    if (st.st_size > 510) {
        fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
        return -1;
    }
    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]);
    return 0;
}
```

一个被系统认为是符合规范的硬盘主引导扇区的特征有以下几点：

- 磁盘主引导扇区只有 512 字节
- 磁盘最后两个字节为 0x55AA
- 由不超过 466 字节的启动代码和不超过 64 字节的硬盘分区表加上两个字节的结束符组成

练习 2：使用 qemu 执行并调试 lab1 中的软件。

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。

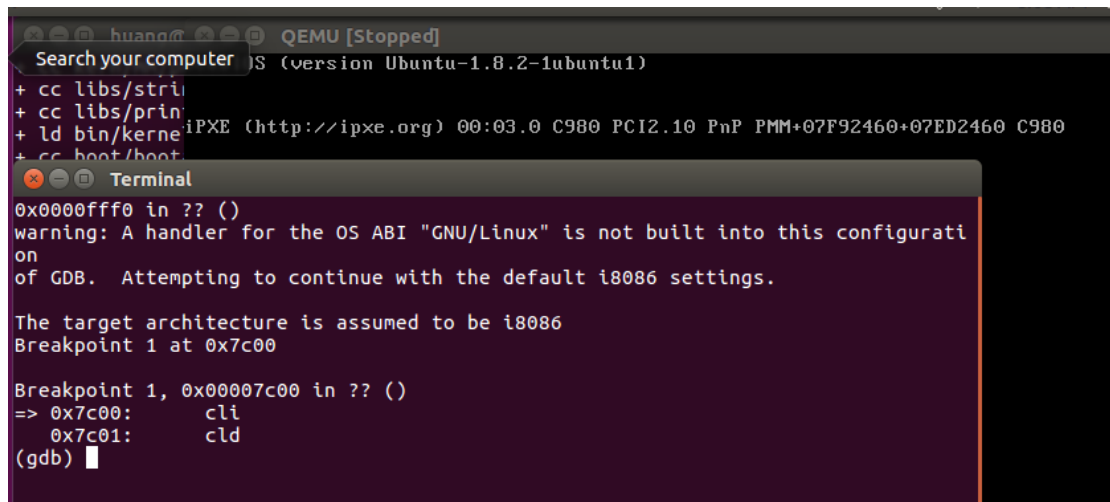
```
gcc: $(TARGETS)
.DEFAULT_GOAL := TARGETS

.PHONY: qemu qemu-nox debug debug-nox
lab1-mon: $(UCOREIMG)
$(V)$(TERMINAL) -e "$(QEMU) -S -s -d in_asm -D $(BINDIR)/q.log -monitor
stdio -hda $< -serial null"
$(V)sleep 2
$(V)$(TERMINAL) -e "gdb -q -x tools/lab1init"
```

```
file bin/kernel      #加载kernel
target remote :1234  #链接qemu
set architecture i8086 #8086的16位实模式
b *0x7c00  #BootLoader第一条指令处设置断点
continue  #继续
x /2i $pc  #输出两行指令
```

2. 在初始化位置 0x7c00 设置实地址断点，测试断点正常。

运行 make lab1-mon, 会在 0x7c00 处中断



```
huang@ ~ % QEMU [Stopped]
Search your computer JS (version Ubuntu-1.8.2-1ubuntu1)
+ cc libs/strli
+ cc libs/prin
+ ld bin/kerne iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980
+ cc boot/boot
Terminal
0x0000fff0 in ?? ()
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
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)
```

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。两者指令相同。

```

Breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:      cli
    0x7c01:      cld
(gdb) x /10i $pc
=> 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) █

```

```

Open ▾  [icon]
# start address should be 0:7c00
bootloader
.globl start
start:
.code16
    cli
    cld

    # Set up the important data
    xorw %ax, %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %ss

    # Enable A20:
    # For backwards compatibility
    # address line 20 is tied low
    # 1MB wrap around to zero bit
seta20.1:
    inb $0x64, %al
empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al
    outb %al, $0x64
port

```

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。

首先我们先分析一下 bootloader:

关闭中断，将各个段寄存器重置，它先将各个寄存器置 0

```
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
```

开启 A20，然后就是将 A20 置 1。当 A20 地址线控制禁止时，程序就像在 8086 中运行，1MB 以上的地址不可访问。而在保护模式下 A20 地址线控制打开，所以需要通过将键盘控制器上的 A20 线置于高电位，使得全部 32 条地址线可用。

```
# 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

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
```

加载 GDT 表

```
# 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.
laddt addt desc
```

将 CRO 的第 0 位置 1


```

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

```

长跳转到 32 位代码段，重装 CS 和 EIP

```

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp $PROT_MODE_CSEG, $protcseg

```

重装 DS、ES 等段寄存器等

```

.code32                                     # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    movw $PROT_MODE_DSEG, %ax             # Our data segment selector
    movw %ax, %ds                         # -> DS: Data Segment
    movw %ax, %es                         # -> ES: Extra Segment
    movw %ax, %fs                         # -> FS
    movw %ax, %gs                         # -> GS
    movw %ax, %ss                         # -> SS: Stack Segment

```

转到保护模式完成，进入 boot 主方法

```

# Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
movl $0x0, %ebp
movl $start, %esp
call bootmain

```

练习 4: 分析 boot loader 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 boot loader 如何加载 ELF 文件。通过分析源代码和通过 qemu 来运行并调试 boot loader&OS，
 boot loader 如何读取硬盘扇区的？
 boot loader 是如何加载 ELF 格式的 OS
 boot loader 读取硬盘扇区

根据上述 bootmain 函数分析，首先是由 readseg 函数读取硬盘扇区，而 readseg 函数则循环调用了真正读取硬盘扇区的函数 readsect 来每次读出一个扇区。读取扇区它用到了 in b

和 out b 这种机器指令。inb 和 outb 的实现都是在 lab0 中介绍到过的内联汇编来实现的，它采取了一种 IO 空间的寻址方式，能够把外设的数据给读到内存中来，这也是一种 X86 里面的寻址方式。Boot loader 把相应的扇区给读进来进去进

一步的分析

```
/* 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);
}
```

bootloader 加载 ELF 格式的 OS

读取完磁盘之后，开始加载 ELF 格式的文件。

```
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 & 0xFFFFF, ph->p_memsz, ph->p_offset);
    }

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

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

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

首先判断是不是 ELF

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

ELF 头部有描述 ELF 文件应加载到内存什么位置的描述表，这里读取出来将之存入 ph

```

struct proghdr *ph, *eph;

// load each program segment (ignores ph flags)
ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);

```

按照程序头表的描述，将 ELF 文件中的数据载入内存

```

for (; ph < eph; ph++) {
    readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
}

```

根据 ELF 头表中的入口信息，找到内核的入口并开始运行

```

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

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。
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。

函数堆栈的原理

理解函数堆栈最重要的两点是：栈的结构，以及 EBP 寄存器的作用。

一个函数调用动作可分解为零到多个 `PUSH` 指令（用于参数入栈）和一个 `CALL` 指令。`CALL` 指令内部暗含了一个将返回地址压栈的动作，由硬件完成。

`print_stackframe` 函数的实现


```

/* 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
 * (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 function's return addr eip = ss:[ebp+4]
 * the calling function's ebp = ss:[ebp]
 */
uint32_t ebp=read_ebp();//(1) call read_ebp() to get the value of ebp. the type is (uint32_t)
uint32_t eip=read_eip();//(2) call read_eip() to get the value of eip. the type is
uint32_t
int i;
for(i=0;i<STACKFRAME_DEPTH&&ebp!=0;i++){//(3) from 0 .. STACKFRAME_DEPTH
    cprintf("ebp:0x%08x eip:0x%08x ",ebp,eip);//(3.1)printf value of ebp, eip
    uint32_t *tmp=(uint32_t *)ebp+2;
    cprintf("arg :0x%08x 0x%08x 0x%08x 0x%08x",*(tmp+0),*(tmp+1),*(tmp+2),*(tmp+3));//
    (3.2)(uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp +2 [0..4]
    cprintf("\n");//(3.3) cprintf("\n");
    print_debuginfo(eip-1);//(3.4) call print_debuginfo(eip-1) to print the C calling
function name and line number, etc.
    eip=((uint32_t *)ebp)[1];
    ebp=((uint32_t *)ebp)[0];//(3.5) popup a calling stackframe
}

```

实验结果

```

huang@ubuntu: ~/Desktop/ucore_os_lab-master/labcodes/lab1
Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x001032a0 (phys)
  edata  0x0010ea16 (phys)
  end    0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b28 eip:0x00100a64 arg :0x00010094 0x00010094 0x00007b58 0x00100092
  kern/debug/kdebug.c:306: print_stackframe+22
ebp:0x00007b38 eip:0x00100d5c arg :0x00000000 0x00000000 0x00007ba8
  kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100092 arg :0x00000000 0x00007b80 0xffff0000 0x00007b84
  kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bc arg :0x00000000 0xffff0000 0x00007ba4 0x00000029
  kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000db arg :0x00000000 0x00100000 0xffff0000 0x0000001d
  kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100101 arg :0x001032bc 0x001032a0 0x0000130a 0x00000000
  kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100055 arg :0x00000000 0x00000000 0x00000000 0x00007c4f
  kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d6a arg :0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d69 --
++ setup timer interrupts

```

最后一行是 ebp:0x00007bf8 eip:0x00007d6a args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8, 共有 ebp, eip 和 args 三类参数, 下面分别给出解释。

ebp:0x00007bf8:

此时 ebp 的值是 kern_init 函数的栈顶地址

eip:0x00007d6e:

eip 的值是 kern_init 函数的返回地址, 也就是 bootmain 函数调用 kern_init 对应的指令的下一条指令的地址。这与 obj/bootblock.asm 相符合。

args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8

一般来说, args 存放的 4 个 dword 是对应 4 个输入参数的值。但这里比较特殊, 由于 bootmain 函数调用 kern_init 并没传递任何输入参数, 并且栈顶的位置恰好在 boot loader 第一条指令存放的地址的上面, 而 args 恰好是 kern_int 的 ebp 寄存器指向的栈顶往上第 2~5 个单元, 因此 args 存放的就是 boot loader 指令的前 16 个字节。

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?

一个表项占八个字节。入口地址为: $gd_off_31_16 \ll 16 + gd_off_15_0$;

```
1 struct gatedesc {
2
3   unsigned gd_off_15_0 : 16; // 低16位为段内偏移
4
5   unsigned gd_ss : 16; // 段选择子占16位
6
7   unsigned gd_args : 5; // # args, 0 for interrupt/trap gates ,also 0
8
9   unsigned gd_rsv1 : 3; // reserved(should be zero I guess)
10
11  unsigned gd_type : 4; // type(STS_{TG,IG32,TG32}) interrupt or trap
12
13  unsigned gd_s : 1; // must be 0 (system)
14
15  unsigned gd_dpl : 2; // descriptor(meaning new) privilege level
16
17  unsigned gd_p : 1; // Present
18
19  unsigned gd_off_31_16 : 16; // 作为高16位的段内偏移。
20
21 };
```

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) {
    /* LAB1 YOUR CODE : STEP 2 */
    /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
     * All ISR's entry addrs are stored in __vectors. where is uintptr_t __vectors[] ?
     * __vectors[] is in kern/trap/vector.S which is produced by tools/vector.c
     * (try "make" command in lab1, then you will find vector.S in kern/trap DIR)
     * You can use "extern uintptr_t __vectors[);" to define this extern variable which
     will be used later.
     * (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).
     * Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to
     setup each item of IDT
     * (3) After setup the contents of IDT, you will let CPU know where is the IDT by using
     'lidt' instruction.
     * You don't know the meaning of this instruction? just google it! and check the
     libs/x86.h to know more.
     * Notice: the argument of lidt is idt_pd. try to find it!
     */
    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);
}

static const char *

```

代码首先引入中断处理函数的入口地址__vectors[],这个变量在 vector.s 里面生成,然后初始化 idt 中断描述符表,最后根据提示用 lidt 函数告知 cpu IDT 表的位置。setgate 这个函数的作用是设置正确的 interrupt/trap gate 描述符。注意需要对 T_SWITCH_TOK 的发生时机是在用户空间的,所以对应的 dpl 需要修改为 DPL_USER。lidt 将 idt 的首地址和 size 装进 idtr 寄存器。

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"。

