Lab1

Question1

1.1 操作系统镜像文件 ucore.img 是如何一步一步生成的?

1.1.1 输入 make V= 得到的结果

```
+ cc kern/init/init.c                          //编译 init.c
gcc -c kern/init/init.c -o obj/kern/init/init.o

+ cc kern/libs/stdio.c                        //编译 stdio.c
    gcc -c kern/libs/stdio.c -o obj/kern/libs/stdio.o

+ cc kern/libs/readline.c                         //编译 readline.c
    gcc -c kern/libs/readline.c -o obj/kern/libs/readline.o

+ cc kern/debug/panic.c                       //编译 panic.c
    Gcc -c kern/debug/panic.c -o obj/kern/debug/panic.o

+ cc kern/debug/kdebug.c                          //编译 kdebug.c
    gcc -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o

+ cc kern/debug/kmonitor.c                    //编译 komnitor.c
    gcc -c kern/debug/kdebug.c -o obj/kern/debug/kmonitor.o

+ cc kern/driver/clock.c                          //编译 clock.c
    gcc -c kern/driver/clock.c -o obj/kern/driver/clock.o

+ cc kern/driver/console.c                    //编译 console.c
gcc -c kern/driver/console -o obj/kern/driver/console.o

+ cc kern/driver/picirq.c                         //编译 prcirq.c
    gcc -c kern/driver/picirq.c -o obj/kern/driver/picirq.o

+ cc kern/driver/intr.c                       //编译 intr.c
    gcc -c kern/driver/intr.c -o obj/kern/driver/intr.o

+ cc kern/trap/trap.c                             //编译 trap.c
```

```
    gcc -c kern/trap/trap.c -o obj/kern/trap/trap.o

+ cc kern/trap/vectors.S                    //编译 vectors.S
    gcc -c kern/trap/vectors.S -o obj/kern/trap/vectors.o

+ cc kern/trap/trapentry.S                  //编译 trapentry.S
    gcc -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o

+ cc kern/mm/pmm.c                          //编译 pmm.c
    gcc -c kern/mm/pmm.c -o obj/kern/mm/pmm.o

+ cc libs/string.c                          //编译 string.c
    gcc -c libs/string.c -o obj/libs/string.o

+ cc libs/printfmt.c                        //编译 printfmt.c
    gcc -c libs/printfmt.c -o obj/libs/printfmt.o

+ ld bin/kernel                             //链接成 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/driverpicirq.o
    obj/kern/driver/intr.o
    obj/kern/trap/trap.o
    obj/kern/trap/vectors.o
    obj/kern/trapentry.o
    obj/kern/mm/pmm.o
    obj/libs/string.o
    obj/libs/printfmt.o

+ cc boot/bootasm.S                         //编译 bootasm.S
    gcc -c boot/bootasm.S -o obj/boot/bootasm.o

+ cc boot/bootmain.c                        //编译 bootmain.c
    gcc -c boot/bootmain.c -o obj/boot/bootmain.o

+ cc tools/sign.c                           //编译 sign.c
    gcc -c tools/sign.c -o obj/sign/tools/sign.o
```

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

+ ld bin/bootblock                          //根据 sign 规范生成 bootblock
    ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00
    obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o

    //创建大小为 10000 个块的 ucore.img，初始化为 0，每个块为 512 字节
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
    //把 bootblock 中的内容写到第一个块
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
记录了 1+0 的读入
记录里 1+0 的写出
    //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
记录了 154+1 的读入
记录了 154+1 的写出
```

1.1.2 生成 libs 目录下的 obj 文件名

listf_cc 函数的定义为 listf_cc = $(call listf,$(1),$(CTYPE)) 可见 listf_cc 又调用了 listf 函数，调用时传入的第 1 个参数为 $(1) = $(LIBDIR) = libs，第 2 个参数为$(CTYPE) = c S。

$(call add_files_cc,$(call listf_cc,$(LIBDIR)),libs,)，当中调用了 add_files_cc，输入的参数有 2 个，第一个参数是调用另外一个函数 listf_cc 的返回值，第二个参数数 libs。

add_files_cc 函数的定义为$(call add_files,$(1),$(cc),$(CFLAGS) $（3）,$(2),$(4))其中调用了函数 add_files。

1.1.3 生成 kern 目录下的 obj 文件名

与上述类似

1.1.4Ucore.img 文件的生成

在 Makefile 文件中发现 ucore.img 文件的生成依赖于 kernel 和 bootblock

```
# 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

$(call create_target,ucore.img)
```

1.1.5 生成 Kernel 的代码

```
# create kernel target
kernel = $(call totarget,kernel)

$(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)

$(call create_target,kernel)
```

1、$(kernel):tools/kernel.ld 指出 kernel 目标文件需要依赖 tools/kernel.ld 文件，而 kernel.ld 文件是一个链接脚本，其中设置了输出的目标文件的入口地址及各个段的一些属性，包括各个段是由输

入文件的哪些段组成、各个段的起始地址等。

2、指出 kernel 目标文件依赖的 obj 文件。

```
$(kernel):$(KOBJS)
KOBJS = $(call read_packet,kernel libs)
```

3、@$(OBJDUMP) -s $@ > $(call asmfile,kernel)使用 OBJDUMP 工具对 kernel 目标文件进行反编码，以便后续进行调试。

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

4、使用 objdump 工具来解析 kernel 目标文件得到符号表

```
        @$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d'
> $(call symfile,kernel)
```

生成 bootblock 的代码

```
# create bootblock
bootfiles = $(call listf_cc,boot)
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -
nostdinc))

bootblock = $(call totarget,bootblock)

$(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)

$(call create_target,bootblock)
```

1、bootfiles = $(call listf_cc,boot) 调用了 listf_cc 函数，listf_cc 函数是过滤出对应目录下的.c 和.S 文件。

2、`bootblock = $(call totarget,bootblock)` 调用了 totarget 函数，totarget 函数是给输入参数增加前缀"bin/"。

3、链接所有.o 文件以生成 obj/bootblock.o：

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

4、反汇编 obj/bootblock.o 文件得到 obj/bootblock.asm 文件：

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

5、使用 objcopy 将 obj/bootblock.o 转换生成 obj/bootblock.out 文件：

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

6、使用 bin/sign 工具将 obj/bootblock.out 转换生成 bin/bootblock 目标文件：

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

生成 ucore.img 的代码

```
# 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

$(call create_target,ucore.img)
```

1、设置了 ucore.img 的目标名：

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

2、 `$(V)dd if=/dev/zero of=$@ count=10000` 为 ucore.img 分配 10000 个 block 的内存空间，并全部初始化为 0。

3、
```
$(V)dd if=$(bootblock) of=$@ conv=notrunc
$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
```

分别将 bootblock 和 kernel 复制到 ucore.img

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?

通过查看 sign.c 文件代码我们可以发现：

```
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;
```

buf 为主引导扇区的缓存,大小为 512 字节，多余的空间用 0 填上。

第 511 个字节是 0x22，第 511 个字节是 0xAA

Question2

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

首先在 lab1_answer 目录下，执行 make debug 命令，即可打开 qemu 和 gdb 并连接。

```
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
kern/init/init.c
12      int kern_init(void) __attribute__((noreturn));
13      void grade_backtrace(void);
14      static void lab1_switch_test(void);
15
        int
17      kern_init(void) {
18          extern char edata[], end[];
19          memset(edata, 0, end - edata);
20
21          cons_init();                // init the console
22
23          const char *message = "(THU.CST) os is loading ...";
24          cprintf("%s\n\n", message);
25
26          print_kerninfo();
emote Thread 1 In: kern_init                                  L17   PC: 0x100000
f GDB.  Attempting to continue with the default i8086 settings.

he target architecture is assumed to be i8086
x0000fff0 in ?? ()
reakpoint 1 at 0x100000: file kern/init/init.c, line 17.

reakpoint 1, kern_init () at kern/init/init.c:17
gdb)
```

执行命令：x/6i $pc

可以看到执行的第一条命令是一条长跳转指令

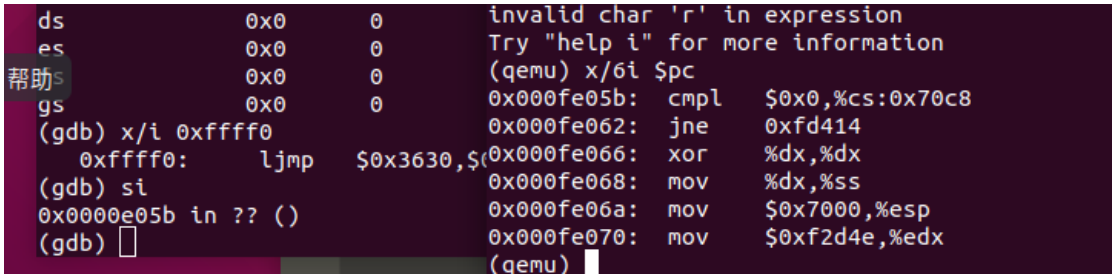
```
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
QEMU 2.11.1 monitor - type 'help' for more information
(qemu) x/6i $pc
0xfffffff0:  ljmp   $0xf000,$0xe05b
0xfffffff5:  xor    %dh,0x322f
0xfffffff9:  xor    (%bx),%bp
0xfffffffb:  cmp    %di,(%bx,%di)
0xfffffffd:  add    %bh,%ah
0xffffffff:  add    %al,(%bx,%si)
(qemu)
```

这里执行命令：i r；查看寄存器状态

可以看到 cs：ip 指向内存开始的地方 0xffff0

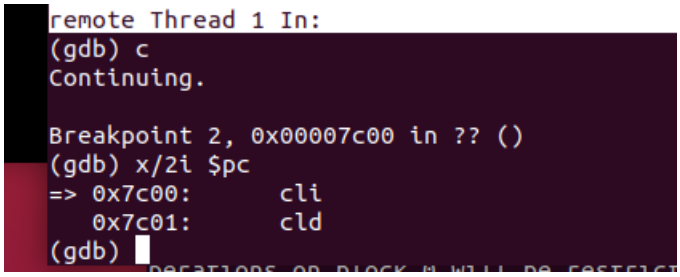
```
eax            0x0      0
ecx            0x0      0
edx            0x663    1635
ebx            0x0      0
esp            0x0      0x0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xfff0   0xfff0
eflags         0x2      [ ]
cs             0xf000   61440
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

继续执行 si 命令，可进行单步的调试工作;
此时再查看指令发现已发生变化



2.2 在初始化位置 0x7c00 设置实地址断点,测试断点正常

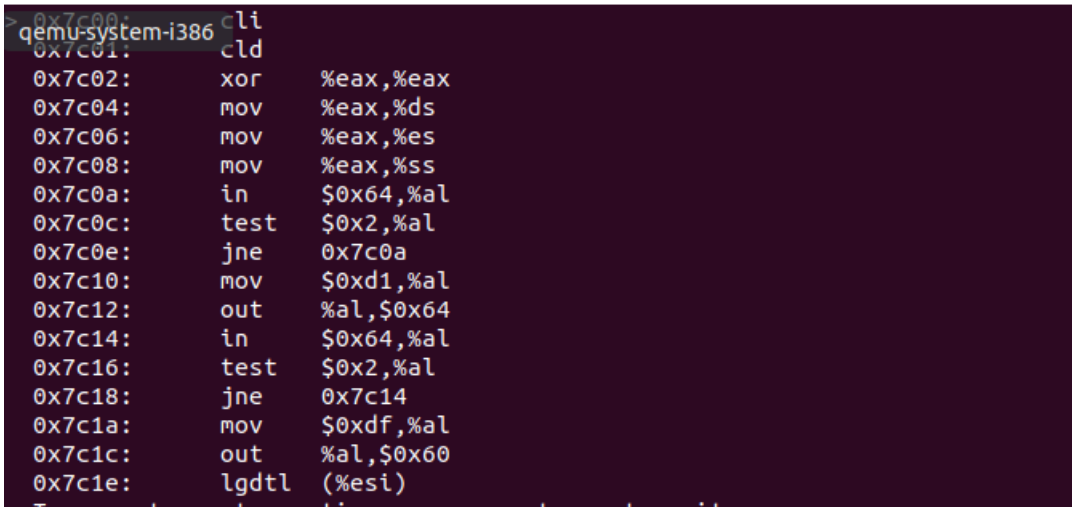
这里通过命令行 b *0x7c00 在此处下断点，之后 c 继续运行，运行到断点处停下后，这里查看代码行如下:



测试正常。

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

这里跟踪得到的代码如下，这里可以看到代码十分简洁，没有多余的东西。



这里是对应的 bootasm.s 的代码，可以看出多了非常多的注释，而且采用的编码方式不太一样，应该在于 x86 和 AT&T 汇编模式的不同。代码段都做了非常充足的解释。

```
.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
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al                                     # 0xd1 -> port 0x64
    outb %al, $0x64                                     # 0xd1 means: write data to 80

seta20.2:
```

而对应的 bootblock.asm 代码则给了更充足的代码段位置，这里与得到的反汇编代码进行比较发现位置是相同的。

```
:ode16                                                  # 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

)007c0a <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.
:ta20.1:
```

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

首先在随便一个运行的位置下断点，在 7c12 处下断点之后执行发现正常。

```
帮助ntinuing.

Breakpoint 2, 0x00007c12 in ?? ()
(gdb) x/2i $pc
=> 0x7c12:      out    %al,$0x64
   0x7c14:      in     $0x64,%al
(gdb)
```

Question3

3.1Bootloader & A20

3.1 bootloader 是如何完成从实模式进入保护模式

下面是在 bootasm.S 文件中 bootloader 进入保护模式的具体代码:
bootloader 入口为 start，根据上个实验的测试，bootloader 会被 BIOS 加载到内存的 0x7c00 处，此时 cs=0, eip=0x7c00，在刚进入 bootloader 的时候，最先执行的操作分别为关闭中断、清除 EFLAGS 的 DF 位以及将 ax, ds, es, ss 寄存器初始化为 0。

```
start:
.code16                                             # Assemble for 16-
bit mode
    cli                                             # Disable interrupt
s
    cld                                             # String operations
 increment
    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number ze
ro
    movw %ax, %ds                                   # -> Data Segment
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment
```

接下来为了使得 CPU 进入保护模式之后能够充分使用 32 位的寻址能力，需要开启 A20，关闭“回卷”机制；该过程主要分为等待 8042 控制器 Inpute Buffer 为空，发送 P2 命令到 Input Buffer, 等待 Input Buffer 为空, 将 P2 得到的第二个位 (A20 选通) 置为 1, 写回 Input Buffer;
接下来对应上述步骤分析 bootasm 中的汇编代码:

首先是从 0x64 内存地址中（映射到 8042 的 status register）中读取 8042 的状态，直到读取到的该字节第二位（input buffer 是否有数据）为 0，此时 input buffer 中无数据;
接下来往 0x64 写入 0xd1 命令，表示修改 8042 的 P2 port;

```
    # 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
```

接下来继续等待 input buffer 为空,往 0x60 端口写入 0xDF, 表示将 P2 port 的第二个位(A20)选通置为 1;

```
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
```

至此，A20 开启，进入保护模式之后可以充分使用 4G 的寻址能力;

接下来需要设置 GDT(全局描述符表), 在 bootasm.S 中已经静态地描述了一个简单的 GDT, 如下所示; 值得注意的是 GDT 中将代码段和数据段的 base 均设置为了 0，而 limit 设置为了 2^32-1 即 4G，此时就使得逻辑地址等于线性地址，方便后续对于内存的操作;

```
# Bootstrap GDT
.p2align 2                                          # force 4 byte alig
nment
gdt:
    SEG_NULLASM                                     # null seg
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)           # code seg for boot
loader and kernel
    SEG_ASM(STA_W, 0x0, 0xffffffff)                 # data seg for boot
loader and kernel

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

因此在完成 A20 开启之后，只需要使用命令 lgdt gdtdesc 即可载入全局描述符表；接下来只需要将 cr0 寄存器的 PE 位置 1，即可从实模式切换到保护模式:

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

接下来则使用一个长跳转指令，将 cs 修改为 32 位段寄存器，以及跳转到 protcseg 这一 32 位代码入口处，此时 CPU 进入 32 位模式；

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

接下来执行的 32 位代码功能为：设置 ds、es, fs, gs, ss 这几个段寄存器，然后初始化栈的 frame pointer 和 stack pointer，然后调用使用 C 语言编写的 bootmain 函数，进行操作系统内核的加载，至此，bootloader 已经完成了从实模式进入到保护模式的任务；

3.2 为何&如何使用 A20

总的来说是本着向后兼容的原则来实现对 4GB 内存的访问

Inter 8088 的 cpu 地址线只有 20 根,其所能寻址的最高地址是 0xffff,也就是 0x10ffef。对于超出 0x100000（1MB）的寻址地址将默认地环绕到 0xffef。Inter 80286 CPU，具有 24 根地址线，最高可寻址 16MB，并且有一个与 8088 那样实现地址寻址的环绕。

但是当时已经有一些程序是利用这种环绕机制进行工作的。为了实现完全的兼容性，IBM 公司发明了使用一个开关来开启或禁止 0x100000 地址比特位。由于当时的 8042 键盘控制器上恰好有空闲的端口引脚（输出端口 P2，引脚 P21），

于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为 A20。如果它为零，则比特 20 及以上地址都被清除。从而实现了兼容性。

当 A20 地址线控制禁止时，程序就像运行在 8086 上，1MB 以上的地址是不可访问的，只能访问奇数 MB 的不连续的地址。为了使能所有地址位的寻址能力，必须向键盘控制器 8082 发送一个命令，键盘控制器 8042 会将 A20 线置于高电位，使全部 32 条地址线可用，实现访问 4GB 内存。

3.3 Gdt 初始化过程

初始化过程在代码底部：

```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment 向后移动位置计数器置为 4 字节的倍数 为了内存对齐
gdt:
  SEG_NULLASM                             # null seg
```

```
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg 可执行段或可读可执行段
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg 可写但是不可执行段

gdtdesc:
  .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1 gdt 表的长度，以字节为单位 这里不太懂为什么这样写
  .long   gdt                             # address gdt .long 后面的参数为 gdt 运行时生成的值，即 gdt 表的地址
```

lgdt gdtdesc

CPU 单独为我们准备了一个寄存器叫做 GDTR 用来保存我们 GDT 在内存中的位置和我们 GDT 的长度。GDTR 寄存器一共 48 位，其中高 32 位用来存储我们的 GDT 在内存中的位置，其余的低 16 位用来存我们的 GDT 有多少个段描述符。 16 位最大可以表示 65536 个数，这里我们把单位换成字节，而一个段描述符是 8 字节，所以 GDT 最多可以有 8192 个段描述符。CPU 不仅用了一个单独的寄存器 GDTR 来存储我们的 GDT，而且还专门提供了一个指令用来让我们把 GDT 的地址和长度传给 GDTR 寄存器：lgdt gdtdesc。

3.4 如何使能和进入保护模式

修改 CR0 寄存器的 PE 值

x86 的控制寄存器一共有 4 个分别是 CR0、CR1、CR2、CR3（这四个寄存器都是 32 位的），而控制进入“保护模式”的开关在 CR0 上。

CR0 中包含了 6 个预定义标志，0 位是保护允许位 PE(Protedted Enable)，用于启动保护模式，如果 PE 位置 1，则保护模式启动，如果 PE=0，则在实模式下运行。

打开保护模式的代码为：

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

因为我们无法直接操作 CR0，所以我们首先要用一个通用寄存器来保存当前 CR0 寄存器的值，这里第一行就是用通用寄存器 eax 来保存 cr0 寄存器的值；

然后 CR0_PE 这个宏的定义在 mmu.h 文件中，是个数值 0x00000001，将这个数值与 eax 中的 cr0 寄存器的值做“或”运算后，就保证将 cr0 的第 0 位设置成了 1 即 PE = 1 保证打开了保护模式的开关。

而 cr0 的第 31 位 PG = 0 表示我们只使用分段式，不使用分页，这时再将新的计算后的 eax 寄存器中的值写回到 cr0 寄存器中就完成了到保护模式的切换。

通过长跳转，更新 CS 寄存器的基地址

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

由于已经使能了保护模式，所以这里要使用逻辑地址，而不是之前实模式的地址了
这里还要注意 PROT_MODE_CSEG 和 PROT_MODE_DSEG，这两者分别定义为 0x8 和 0x10，表示代码段和数据段的选择。

设置段寄存器，并建立堆栈

这里建立堆栈，ebp 寄存器按理来说是栈帧的，但是这里并不需要把它设置为 0x7c00，因为这里 0x7c00 是栈的最高地址，它上面没有有效内容，而之后因为调用，ebp 会被设置为被调用的那个函数的栈的起始地址，这里就不用管它了。

```
1         movw $PROT_MODE_DSEG, %ax
2         movw %ax, %ds
3         movw %ax, %es
4         movw %ax, %fs
5         movw %ax, %gs
6         movw %ax, %ss
7         movl $0x0, %ebp
8         movl $start, %esp
```

练习 4:

实验内容:

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?

相关知识点：

ELF：(Executable and linking format)文件格式是 Linux 系统下的一种常用目标文件(object file)格式，有三种主要类型:

用于执行的可执行文件(executable file)，用于提供程序的进程映像，加载到内存执行。 这也是本实验的 OS 文件类型。

用于连接的可重定位文件(relocatable file)，可与其它目标文件一起创建可执行文件和共享目标文件。

共享目标文件(shared object file),连接器可将它与其它可重定位文件和共享目标文件连接成其它的目标文件,动态连接器又可将它与可执行文件和其它共享目标文件结合起来创建一个进程映像。

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

/* ***********************************************************************
*
 * This a dirt simple boot loader, whose sole job is to boot
 * an ELF kernel image from the first IDE hard disk.
 *
 * DISK LAYOUT
 *  * This program(bootasm.S and bootmain.c) is the bootloader.
 *    It should be stored in the first sector of the disk.
 *
 *  * The 2nd sector onward holds the kernel image.
 *
 *  * The kernel image must be in ELF format.
 *
 * BOOT UP STEPS
 *  * when the CPU boots it loads the BIOS into memory and executes it
 *
 *  * the BIOS intializes devices, sets of the interrupt routines, and
 *    reads the first sector of the boot device(e.g., hard-drive)
 *    into memory and jumps to it.
 *
 *  * Assuming this boot loader is stored in the first sector of the
 *    hard-drive, this code takes over...
 *
 *  * control starts in bootasm.S -- which sets up protected mode,
 *    and a stack so C code then run, then calls bootmain()
```

```
 *
 *  * bootmain() in this file takes over, reads in the kernel and jumps
 to it.
 * */

#define SECTSIZE        512
#define ELFHDR          ((struct elfhdr *)0x10000)      // scratch spac
e

/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}

/* 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);
}

/* *
 * readseg - read @count bytes at @offset from kernel into virtual addr
ess @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);
    }
}

/* 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))();

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

    /* do nothing */
```

```
    while (1);
}
```

Bootmain 的主要内容：

bootasm.S 完成了 bootloader 的大部分功能，包括打开 A20，初始化 GDT，进入保护模式，更新段寄存器的值，建立堆栈，接下来 bootmain 完成 bootloader 剩余的工作，就是把内核从硬盘加载到内存中来，并把控制权交给内核。

问题 1：bootloader 如何读取硬盘扇区的？

读硬盘扇区的代码如下:

```
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);
}
```

就是把硬盘上的 kernel，读取到内存中

从 outb()可以看出这里是用 LBA 模式的 PIO（Program IO）方式来访问硬盘的（即所有的 IO 操作是通过 CPU 访问硬盘的 IO 地址寄存器完成）。从磁盘 IO 地址和对应功能表可以看出，该函数一次只读取一个扇区。

其中 insl 的实现如下：

```
static inline void
insl(uint32_t port, void *addr, int cnt) {
    asm volatile (
            "cld;"
             "repne; insl;"
            : "=D" (addr), "=c" (cnt)
            : "d" (port), "0" (addr), "1" (cnt)
            : "memory", "cc");
}
```

读取硬盘扇区的步骤：

1. 等待硬盘空闲。waitdisk 的函数实现只有一行：while ((inb(0x1F7) & 0xC0) != 0x40)，意思是不断查询读 0x1F7 寄存器的最高两位，直到最高位为 0、次高位为 1（这个状态应该意味着磁盘空闲）才返回。

2. 硬盘空闲后，发出读取扇区的命令。对应的命令字为 0x20，放在 0x1F7 寄存器中；读取的扇区数为 1，放在 0x1F2 寄存器中；读取的扇区起始编号共 28 位，分成 4 部分依次放在 0x1F3~0x1F6 寄存器中。

3. 发出命令后，再次等待硬盘空闲。

4. 硬盘再次空闲后，开始从 0x1F0 寄存器中读数据。注意 insl 的作用是“That function will read cnt dwords from the input port specified by port into the supplied output array addr.”，是以 dword 即 4 字节为单位的，因此这里 SECTIZE 需要除以 4.

问题二：bootloader 如何加载 ELF 格式的 OS

1. 从硬盘读了 8 个扇区数据到内存 0x10000 处，并把这里强制转换成 elfhdr 使用；

2. 校验 e_magic 字段；

3. 根据偏移量分别把程序段的数据读取到内存中。

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) {
    // wait for disk to be ready
    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);                      // cmd 0x20 - read sectors

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

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

    // 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.
    //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) {
    // read the 1st page off disk 读取ELF的头部
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF? 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;

    // load each program segment (ignores ph flags)
    // 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);
    }

    // call the entry point from the ELF header ELF文件0x1000位置后面的0xd1ec比特被载入内存0x00100000
    // note: does not return 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);

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

练习五

代码书写过程说明:

根据注释一步一步完成即可。

1、 首先定义两个局部变量 ebp 和 eip 来存放寄存器 ebp 和 eip 的值。

2、 调用 read_ebp 函数来获取执行 print_stackframe 函数时 ebp 寄存器的值。

3、 调用 read_eip 函数来获取当前指令的位置，eip 寄存器的值。

4、 由于变量 eip 存放的是下一条指令的地址，因此将变量 eip 的

值减去 1，得到的指令地址就属于当前指令的范围。

```
uint32_t ebp=read_ebp();
uint32_t eip=read_eip();
int i;
for(i=0;i<STACKFRAME_DEPTH&&ebp!=0;i++)
{
  cprintf("ebp:0x%08x eip:0x%08x ",ebp,eip);
  uint32_t *args=(uint32_t *)ebp+2;
  cprintf("arg :0x%08x 0x%08x 0x%08x 0x%08x\n",*(args+0),*
(args+1),*(args+2),*(args+3));

  print_debuginfo(eip-1);
  eip=((uint32_t *)ebp+1);
  ebp=((uint32_t *)ebp+0);
}
```

```
ebp:0x00007b28 eip:0x00100ab4 arg :0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:306: print_stackframe+26
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
```

最后一行：共有 ebp，eip 和 args 三类参数

```
ebp:0x00007b28 eip:0x00007b2c arg :0x00010094 0x00010094 0x00007b58 0x00100096
    <unknow>: -- 0x00007b2b --
```

ebp:0x00007b28 ebp 的值是 kern_init 函数的栈顶地址。

eip:0x0007b2c 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。

args:0x00010094 0x00010094 0x00007b58 0x00100096 args 存放的就是 boot loader 指令的前 16 个字节

```
start:
.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
```

实验 6:

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”。

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？

一个表项的结构如下:

```
/*Lab1/kern/mm/mmu.h*/
/* 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
};
```

中断描述符表一个表项占 8 个字节，其结构如下：

bit 63…48: offset 31…16

bit 47…32: 属性信息，包括 DPL、P flag 等

bit 31…16: Segment selector

bit 15…0: offset 15…0

其中第 16 ~ 32 位是段选择子，用于索引全局描述符表 GDT 来获取中断处理代码对应的段地址，再加上第 0 ~ 15、48 ~ 63 位构成的偏移地址，即可得到中断处理代码的入口。

问题 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

idt_init 函数的功能是初始化 IDT 表。IDT 表中每个元素均为门描述符，记录一个中断向量的属性，包括中断向量对应的中断处理函数的段选择子/偏移量、门类型（是中断门还是陷阱门）、DPL 等。因此，初始化 IDT 表实际上是初始化每个中断向量的这些属性。

题目已经提供中断向量的门类型和 DPL 的设置方法：除了系统调用的门类型为陷阱门、DPL=3 外，其他中断的门类型均为中断门、DPL 均为 0.

中断处理函数的段选择子及偏移量的设置要参考 kern/trap/vectors.S 文件：由该文件可知，所有中断向量的中断处理函数地址均保存在__vectors 数组中，该数组中第 i 个元素对应第 i 个中断向量的中断处理函数地址。而且由文件开头可知，中断处理函数属于.text 的内容。因此，中断处理函数的段选择子即.text 的段选择子 GD_KTEXT。从 kern/mm/pmm.c 可知.text 的段基址为 0，因此中断处理函数地址的偏移量等于其地址本身。

完成 IDT 表的初始化后，还要使用 lidt 命令将 IDT 表的起始地址加载到 IDTR 寄存器中。

```
/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S */

void idt_init(void) {

    extern uintptr_t __vectors[];  //保存在 vectors.S 中的 256 个中断处理例程的入口地址数组

    int i;

    /* along: how to set istrap and dpl? */

   //使用 SETGATE 宏，对中断描述符表中的每一个表项进行设置

    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) { //IDT 表项的个数

    //在中断门描述符表中通过建立中断门描述符，其中存储了中断处理例程的代码段 GD_KTEXT 和偏移量__vectors[i]，特权级为 DPL_KERNEL。这样通过查询 idt[i]就可定位到中断服务例程的起始地址。

        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);

    }
```

```
SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT,
    __vectors[T_SWITCH_TOK], DPL_USER);
```

//建立好中断门描述符表后，通过指令 lidt 把中断门描述符表的起始地址装入 IDTR 寄存器中，从而完成中段描述符表的初始化工作。

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

上述代码中 SETGATE 函数的实现：

```
#define SETGATE(gate, istrap, sel, off, dpl) {            \
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;        \
    (gate).gd_ss = (sel);                                 \
    (gate).gd_args = 0;                                    \
    (gate).gd_rsv1 = 0;                                    \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;    \
    (gate).gd_s = 0;                                      \
    (gate).gd_dpl = (dpl);                                 \
    (gate).gd_p = 1;                                      \
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16;          \
}
```

宏定义和数组说明：

```
#define GD_KTEXT    ((SEG_KTEXT) << 3)          // kernel text
#define DPL_KERNEL    (0)
#define DPL_USER    (3)
#define T_SWITCH_TOK                121    // user/kernel switch
static struct gatedesc idt[256] = {{0}};
```

问题 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

trap 函数只是直接调用了 trap_dispatch 函数，而 trap_dispatch 函数实现对各种中断的处理，题目要求我们完成对时钟中断的处理，实现非常简单：定义一个全局变量 ticks，每次时钟中断将 ticks 加 1，加到 100 后打印"100 ticks"，然后将 ticks 清零重新计数。代码实现如下：

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