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。

答：

bootloader 工作的实现文件在 lab1/boot 目录下，包含以下三个文件：

- bootasm.S —— 定义并实现了 bootloader 最先执行的函数 `start`，此函数进行了一定的初始化，完成了从实模式到保护模式的转换，并调用 bootmain.c 中的 `bootmain` 函数；
- bootmain.c —— 定义并实现了 `bootmain` 函数，实现了通过屏幕、串口和并口显示字符串。`bootmain` 函数加载 uCore 操作系统到内存，然后跳转到 uCore 的入口处执行；
- asm.h —— 是 bootasm.S 汇编文件所需要的头文件，主要是一些与 x86 保护模式的段访问方式相关的宏定义。

经过练习 2 可以知道，CPU 加电后执行的 0x7c00 处的代码即为 bootasm.S 中的代码，因此该文件中的代码被最先执行。阅读代码的源文件，可以看到，在代码的第 23 行之前，进行了一些初始化置 0 的操作，如下图所示：

```
12  # 在实模式中，bootloader的起始执行地址应位于0:7c00
13  .globl start
14  start:
15  .code16                                             # 16位实模式
16      cli                                             # 屏蔽系统中断（设置IF为0）
17      cld                                             # 设置方向标志位DF为0
18
19      # Set up the important data segment registers (DS, ES, SS).
20      xorw %ax, %ax                                   # ax置0
21      movw %ax, %ds                                   # -> 数据段
22      movw %ax, %es                                   # -> 附加段
23      movw %ax, %ss                                   # -> 栈段
```

实模式下（16 位模式），只能访问 1MB 物理内存空间。必须通过修改 A20 地址线（就是指地址总线中的第 20 根）才能从实模式转换到保护模式。这里涉及到早期 CPU 为了节省硬件设计成本而使用 8042 键盘控制器来控制 A20 Gate。

```
25      # 开启A20:
26      #   为了兼容早期的PC，一开始时A20地址线是被屏蔽的（总
27      #   为0），因此，超过1M的内存被默认“回卷”到0地址处，
28      #   下面这段代码用于使能A20。
29  seta20.1:
30      inb $0x64, %al                                  # 等待8042 Input Buffer为空
31      testb $0x2, %al
32      jnz seta20.1
33
34      movb $0xd1, %al                                 # 写入0xd1 -> 0x64端口
35      outb %al, $0x64                                 # 0xd1的含义是：写数据到8042的P2端口
36
37  seta20.2:
38      inb $0x64, %al                                  # 等待8042 Input Buffer为空
39      testb $0x2, %al
40      jnz seta20.2
41
42      movb $0xdf, %al                                 # 写入0xdf -> 0x60端口
43      outb %al, $0x60                                 # 0xdf = 11011111，含义是设置P2的A20位（第1位）为1
```

这段代码先从 8042 Input Buffer 中读取数据，使用 `testb` 判断它的第 2 位（索引为 1）是否为 1，如果为 1 说明缓冲器中还有数据没有被处理，直到缓冲器空，继续写数据 `0xd1`（`11010001`）到 `0x64` 端口，发送一个键盘控制命令，从而禁止键盘的操作。然后再次等待直到缓冲器空，写数据 `0xdf`（`11011111`）到 `0x60` 端口，即设置 P2 的 A20 位为 1，自此，全部 32 条地址线均可以使用，能够访问 4G 的内存空间。

接下来需要将 CPU 从实模式切换到保护模式，在此之前先使用 lgdt 命令载入已经存在的 GDT 全局描述符表，并将 GDT 表的首地址加载到 GDTR：

```
45      # 从实模式切换到保护模式，同时使用一个可以将虚拟
46      # 内存地址映射到实际内存地址的GDT表和段选择子，
47      # 并保证起作用的内存地址映射不会发生改变。
48      lgdt gdtdesc
```

其中 `gdtdesc` 是在文件的末尾定义的：

```
76  # 全局描述符表GDT
77  .p2align 2                                          # 强制4字节对齐
78  gdt:
79      SEG_NULLASM                                     # 空段
80      SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)           # bootloader和kernel的代码段
81      SEG_ASM(STA_W, 0x0, 0xffffffff)                 # bootloader和kernel的数据段
82
83  gdtdesc:
84      .word 0x17                                      # sizeof(gdt) - 1
85      .long gdt                                       # GDT起始地址
86
```

可以看出 GDT 表的存放位置是 4 字节对齐的，即 GDT 表的物理首地址是 4 的整数倍，`gdt` 部分标识了 3 个 GDT 表项，他们使用了 asm.h 中的两个宏 `SEG_NULLASM` 和 `SEG_ASM`，其中 SEG_NULLASM 的定义如下：

```
7  #define SEG_NULLASM                                     \
8      .word 0, 0;                                         \
9      .byte 0, 0, 0, 0
```

这里定义了 8 个字节为 0，是一个空的 GDT 表项。

SEG_ASM 宏的定义为：

```
11 #define SEG_ASM(type,base,lim)                         \
12     .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);  \
13     .byte (((base) >> 16) & 0xff), (0x90 | (type)),     \
14         (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

在这里，type 表示段属性，base 表示段基址，lim 则表示段长的界限，给出这三个参数就可以用这个宏来定义一个 GDT 表项。

其中的 type 也是宏定义的：

```
18  #define STA_X       0x8     // 可执行的段
19  #define STA_E       0x4     // 向下扩展（仅用于不可执行的段）
20  #define STA_C       0x4     // 一致性的代码段（仅用于可执行的段）
21  #define STA_W       0x2     // 可写（仅用于不可执行的段）
22  #define STA_R       0x2     // 可读（仅用于可执行的段）
23  #define STA_A       0x1     // 可访问
```

多个不同的 type 使用按位或（|）连接。

进入保护模式之前的所有工作到此都已经处理完毕，接下来就是设置控制寄存器 CR_0 上的 PE 位为 1 以便使 CPU 进入保护模式：

```
49      movl %cr0, %eax
50      orl $CR0_PE_ON, %eax
51      movl %eax, %cr0
```

这里 CR0_PE_ON 的值定义为 1，通过 orl 这个 32 位的按位或指令利用 eax 寄存器间接将寄存器 CR0 上的第 0 位使能设置为 1。

之后，用一个跳转指令让系统开始使用 32 位的寻址模式：

```
53      # 跳到下一条指令，进入32位的代码段
54      # 切换处理器至32位保护模式
55      ljmp $PROT_MODE_CSEG, $protcseg
```

可以看到第 55 行的长跳转指令实际上是在系统进入保护模式后执行的。于是在这里 $PROT_MODE_CSEG，代表的是段选择子，从前面的 GDT 表中可以看到基地址是 0x0，而偏移地址是$protcseg， $protcseg 实际上代表的是接下来指令的链接地址，也就是可执行程序在内存中的虚拟地址，只是刚好在这里编译生成的可执行程序 boot 的加载地址与

链接地址是一致的，于是$protcseg 就相当于指令在内存中存放位置的物理地址，所以这个长跳转可以成功的跳转到下一条指令的位置。

ljmp 指令实际上改变了代码段寄存器%cs 的值，让它指向了 GDT 中的一个代码描述符表项，将处理器真正切换到 32 位模式。最后，初始化除代码段外的段寄存器为 0x10，然后设置堆栈指针指向 start 处，然后执行 call bootmain 调用 C 函数：

```
57  .code32                                             # 32位保护模式
58  protcseg:
59      # 初始化保护模式下的数据段寄存器
60      movw $PROT_MODE_DSEG, %ax                       # 数据段选择子
61      movw %ax, %ds                                   # -> DS: 数据段
62      movw %ax, %es                                   # -> ES: 附加段
63      movw %ax, %fs                                   # -> FS
64      movw %ax, %gs                                   # -> GS
65      movw %ax, %ss                                   # -> SS: 栈段
66
67      # 设置堆栈指针并调用C程序。堆栈区域从0到start（0x7c00）
68      movl $0x0, %ebp
69      movl $start, %esp
70      call bootmain
71
72      # 如果bootmain返回了(它不应该返回)，则一直循环。
73  spin:
74      jmp spin
```

实际上这个函数永远都不会返回，末尾 73-74 行的无限循环实际上并不会被执行到。

附录 1：

本练习使用到的部分 AT&T 汇编指令含义

汇编指令	含义
xorw	16 位按位或运算
movw	16 位字传送
movb	8 位字节传送
movl	32 位双字传送
inb	向 I/O 端口写入一个字节
outb	从 I/O 端口读取一个字节
orl	32 位按位或运算
testb	字节的与运算，仅修改标志位，不返回结果
lgdt	装载全局描述符表 GDT 到全局描述符表寄存器 GDTR

参考资料：

[1] 自在随心. uCore 实验 1 笔记整理[OL]. http://qiaoin.github.io/ucore-ex1-notes.html.