代码阅读报告（一）：引导区（必选专题）

（1） 关键代码分析与注释

主要分析 bootasm.S 文件与 bootmain.c 文件。

在 bootasm.S 文件中，首先出现的是中断禁用指令 cli（1015 行），该指令禁用系统中断：因为此时 BIOS 刚刚运行完毕，而它在运行的时候可能已经初始化了中断句柄，所以此时系统是能够接受中断指令的，但是 BIOS 已经停止工作，而真正的操作系统（xv6）此时还没开始运行，所以此时如果接受中断指令，会引发意外的情况，所以要禁用中断，在 xv6 初始化完成后再启用中断。

然后初始化若干寄存器，1018 行的“xorw %ax,%ax ”通过异或的位运算实现 ax 寄存器置零，然后用 mov 指令将 ds,es,ss 置零。

之后是启动第 21 个地址位的操作。由于处理器之前在模拟 8088（只有 20 位），此处需要通过控制键盘控制器的输出端口（0x64,0x60）来启用，1027-1030 行是一个小循环，直到 0x64 的值变为 0x2，才跳出循环，然后把 0x64 端口的值置为 0xd1。1035-1041 行与上面同理，目的是把 0x60 端口的值置为 0xdf。

下一步需要将 16 位模式转换到 32 位模式，建立 GDT 表（1054-1057），然后将 cr0 置为 CR0_PE，直到 1063 行的长跳转指令结束后，系统才真正切换到 32 位模式。

1067-1074 行，开始 32 位模式最初的各个寄存器初始化。此时虚拟地址已经映射到真实地址，准备开始运行 bootmain.c。

1077 行，为操作系统分配栈（esp 即为栈指针），1078 行，调用 bootmain

之后是异常处理（bootmain 返回了值），并将异常结果返回到 0x8a00 端口（如果操作系统运行在实体机，这个端口没有响应，如果运行在 Bochs 上，这个端口会引发 Bochs 的 debug）。

下面分析 bootmain.c 文件。

1223 行，为 elfheader 预留缓冲区。

1226 行，开始调用函数 readseg，在函数 readseg 中，以扇区为单位，逐一执行 readsect 函数（1260 行），注意此时需要将字节数转换为扇区数，这里是除以 512（1280 行），因为一个扇区是 512 字节。读取首个 elf 文件并返回指向该文件的指针。

1229 行，检查“魔数”，以确认该文件是所需的 elf 文件，之后从 1233 行开始，逐一读取各个 elf 文件（这里不同的 elf 文件以指针相连，结构类似于单链表）。

各个扇区读取完毕后（也就是各个命令被加载到对应的地址后），内核指针指向 main（1244-1245 行），系统加载完毕。

（2） 操作系统引导流程分析

BIOS 启动完毕，开始运行 bootasm.S，为防止异常，在加载真正的操作系统（xv6）之前禁用中断。然后将高位寄存器置零，加载 GDT 表，准备进入 32 位的保护模式。经过一个长跳转指令，进入 32 位保护模式，然后进行一系列的寄存器初始化，加载栈指针，调用 bootmain.c 函数。

如果调用成功，在 bootmain.c 中，建立缓冲区，然后开始读取操作系统的第一页。检查魔数，如果成功，则依次读取操作系统的所有页。在读取每一个页的过程中，以扇区为单位逐一读取页的全部内容。

如果调用失败，bootmain.c 返回一个值，交给 bootasm.S 处理，若系统运行于真实的计算机上，则没有反应，若运行在 Bochs 上，则引发 Bochs 的 debug 操作。

（3） 对于简答题目的回答

1. xv6 映像的实现

首先，在 makefile 开始的位置用宏定义实现了几个工具，然后，在下面这段代码：

```
bootblock: bootasm.S bootmain.c
  $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
  $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o
    bootasm.o bootmain.o
  $(OBJDUMP) -S bootblock.o > bootblock.asm
  $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
  ./sign.pl bootblock
```

使用 bootasm.S 以及 bootmain.c 生成了 bootblock.o 文件，然后使用刚才定义的 objcopy 工具复制 bootblock.o 的.text 段（也就是程序的可执行指令）以生成 bootblock。然后，使用 perl 脚本的 sign.pl 把 bootblock 设置成 512 长度（空白部分置零，最后 2 个字节分别为\x55\xAA），得到的就是 bootloader，在 BIOS 自检结束后，首先执行的就是 bootloader。同时，该段生成的 entryother 和 initcode 也只包含可执行指令。

然后是下面这段代码：

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld
$(LD) $(LDFLAGS) -T kernel.ld -o kernel entry.o $(OBJS) -b binary initcode entryother
$(OBJDUMP) -S kernel > kernel.asm
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > kernel.sym
```

按照连接脚本 kernel.ld 的格式，把 entry.o $(OBJS)等文件连接，生成 kernel。同时，把 binary initcode entryother 放入到 kernel 中，生成相应的全局变量_bianry_*_start 和_binary_*_size，以便于在程序中定位指令的位置及大小。

最后是这段代码：

```
xv6.img: bootblock kernel fs.img
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

使用 dd 工具，把 bootblock 放在第一个大小为 512 的块（BIOS 自检结束后，直接加载此块），kernel 放在第一个块之后，生成 xv6.img

2．xv6 是如何做准备并进入保护模式的

首先，禁用中断；然后，设置对应端口，开启 A20 地址线；第三步，建立 GDT 表，将 CR0_PE 装入寄存器 cr0 开启保护模式。最后，将三个高位寄存器初始化为 0x10，使虚拟内存与物理内存匹配。

3. 引导程序是如何读取硬盘扇区的？又是如何加载 ELF 格式的 OS 的？

在 bootmain 中，引导程序调用 redseg()读取硬盘中从 offset 开始的大小为 count 字节的数据所在的所有扇区的数据。在 readseg 函数中，系统通过计算 offset=(offset/SECTSIZE)+1 获得第一个扇区的 LBA 参数，然后在 for 循环中调用函数 readsect，读取每一个扇区。在 readsect 函数中，调用 outb 过程将读取的扇区数目传给 0x1F2 寄存器，将扇区 LBA 参数传给硬盘的 0x1F3-0x1F6 寄存器，之后将 0x1F6 的高四位置为 1110，第六位为 1 表示硬盘读取

方式为 LBA，第四位为 0 表示从主盘读取数据。最后，将读取命令（0x20 处）发送给硬盘的状态命令寄存器 0x1F7。待硬盘就绪后，从数据寄存器(0x1F0)处读取指定扇区的数据。扇区读取的具体实现，是代码 cld　reo insl

下面加载 ELF 格式的 OS。ELF 头保存在硬盘的第一个扇区中，系统通过读取硬盘扇区将 ELF 头的数据加载到从 0x10000 开始的内存段中。检查魔数，从而检查 elfhdr 合法性后，通过 elfhdr 中保存的 prohdr 数组信息将内核启动所需的段加载到内存中。最后，系统通过 elfhdr 中保存的 entry 信息跳转到主程序入口，OS 加载完成。

4. 阅读心得

对于硬件发展的历史有了初步的了解。在研究代码中 A20 地址线的操作的过程中，查阅了较多资料，知道了为什么会这样做。同时也了解了 IDE 磁盘以及相关工程标准。另外初步学习了汇编语言的 AT&T 标准，虽然与 Intel 标准相似，但两者之间有很多不同（不局限于源操作数与目的操作数的先后顺序上）。

隋唯一 软件 71 2017011430