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。

1. 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。

1. 对于简答题目的回答
2. 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。

1. 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。

1. 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