代码阅读报告1

操作系统启动引导的流程分析

1. 首先，BIOS完成硬件自检等工作之后将控制权交给 `bootasm.S` 文件中的 `start` 入口；
2. 进入 `bootasm.S` 之后第一件事情就是将处理器中断禁用，因为此时系统没有开始工作，开启中断是危险的。接着将三个段寄存器清零，为后面的工作做好初始化；

```
.code16                       # Assemble for 16-bit mode
.globl start
start:
 # disable processor interrrupts
 # because BIOS stops working and it's dangerous to enable interrupts
 # when xv6 is ready, interrupts will be enabled again

  cli                         # BIOS enabled interrupts; disable

 # now xv6 is in "real model", simulates an Intel 8088
 # about segment register(16 bits):
 # instruxtion fetches use %cs
 # data reads and writes use %ds
 # stack reads and writes use %ss

 # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

3. 接着，为了使用超过20位的地址线，需要打开A20端口（由IBM规定的键盘端口0x64，0x60）；

```
# Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.

# open A20 gate to enable the 21st address bit
# the A20 gate is controlled by kbd port 0x64 and 0x60
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

4. 下面要从实模式进入保护模式，此时要先设置全局描述表，通过 `lgdt` 命令将处理器的全局描述表指针指向定义好的 `gdtdesc` ，该表中有三个表项，一个为空项，一个为代码段，一个是数据段，其中后两个均定义在0到4G的地址；接着通过预定义的 `CR0_PE` 给 `%cr0` 赋值，打开保护模式；

```
# Switch from real to protected mode.  Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# effective memory map doesn't change during the transition.

# start to use a predefined GDT table
# the GDT table has a null entry
# one entry for executable code
# one entry for data
# lgdt sets the processor's GDT register with the value gdtdesc
# enable protected mode by setting the 1 bit (CR0_PE) in register %cr0
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0
```

5. 虽然已经打开了保护模式的开关，但是此时仍然是16位寻址模式，所以需要使用 `ljmp` 命令重新给 `%cs` 与 `%eip` 寄存器赋值，使之指向32位的指令开头 `start32` ；

```
# use long jump to reload %cs -> SEG_KCODE<<3 and %eip -> start32
  ljmp    $(SEG_KCODE<<3), $start32
```

6. 进入保护模式之后，首先给所有的段寄存器重新赋值，其中 `%ds`，`%es`，`%ss` 赋值为 `0x0008`，也即使得段寄存器的高13位为1，使之指向GDT表中第一项，使之能够在32位模式下正常工作；到现在为止已经完成了从16位实模式向32位保护模式转化的工作，但是现在仍然无法执行C代码，因为此时还没有设立栈；于是接下来就将 `start` 所指向的地址 `0x7c00` 一直到 `0x0` 分配为内核栈，将栈指针 `%esp` 指向 `start`；至此 `bootasm.S` 的工作基本完成，调用 `bootmain` 函数将控制权转交给 `bootmain.c`；

```
.code32  # Tell assembler to generate 32-bit code now.
start32:
# Set up the protected-mode data segment registers

# reset 3 segment(%ds,%es,%ss) registers to SEG_KDATA<<3
# set 2 segments(%fs,%gs) that are not ready to use to 0
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

# Set up the stack pointer and call into C.
# $start=0x7c00, the stack is placed from $start to 0
  movl    $start, %esp
  call    bootmain
```

7. 进入 `bootmain` 函数后，主要工作就是将内核代码载入内存中从 `0x10000` 开始的连续空间；为了将内核代码加载进来，主要有如下的几个步骤；

8. 首先先加载头4096个字节的内容，检查文件头是否满足ELF文件格式，如果满足则继续，否则返回，将控制权转交给 `bootasm.S`，向端口 `0x8a00` 输出错误信息后陷入死循环；

```
  struct elfhdr *elf; //pointer to elf struct
  struct proghdr *ph, *eph; //pointers to the start and end of program headers
  void (*entry)(void); //function pointer to entry()
  uchar* pa; //segment pointer

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error
```

9. 接下来由于ELF内部文件是分段组织的，因此采取分段读入的循环；在每一段的读入中，由于每一段也是由若干 `sectors` 组成，因此采取分 `sector` 的读入循环；读入每一个 `sector` 时，需要向IDE磁盘请求数据，请求的步骤如下：首先等待磁盘准备好接受请求，通过向几个端口写数据向磁盘传达需要取出的数据的位置信息，等待磁盘完成读出数据到缓存中，从特定的端口取回数据；

```
// Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum; //point to the end of program segments
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off); //from pa(program address)+phoff read fi
lesz data
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz); //fill the rest of me
msz with 0
  }
```

```
void
waitdisk(void)
{
  // Wait for disk ready.
  while((inb(0x1F7) & 0xC0) != 0x40)
    ;
}

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  // pass variables to IDE disk
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  // tell the IDE disk to read data given variables
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  // after disk reading the data, read all the data from disk port
  insl(0x1F0, dst, SECTSIZE/4);
}

// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
```

```
// Might copy more than asked.
void
readseg(uchar* pa, uint count, uint offset)
{
  uchar* epa;

  // mark the end of the segment
  epa = pa + count;

  // Round down to sector boundary.
  pa -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (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.
  // read a sector at a time
  for(; pa < epa; pa += SECTSIZE, offset++)
    readsect(pa, offset);
}
```

10. 将内核代码全部加载完成之后，调用ELF预先定义好的入口函数 `entry()` ，此时 `bootmain.c` 就完成了它的工作；操作系统的启动引导就完成了。

```
  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
```

第三部分简答题的简要回答

1. 问题：仔细阅读 `Makefile` ，分析 `xv6.img` 是如何一步一步生成的?

   答： `Makefile` 描述了可执行文件之间的依赖关系，因此可以采取自顶向下的分析方式来分析 `xv6.img` 是如何生成的。首先 `xv6.img` 的生成规则中给出了该文件的依赖文件为 `bootblock` , `kernel` , `fs.img` ，再将这三个文件按照一定的地址关系复制生成了 `xv6.img` ；下面考察 `bootblock` 的生成过程， `bootblock` 依赖于两个源文件 `bootasm.S` , `bootmain.c` ，通过 `gcc` 工具将上述源文件编译后，使用 `ld` 工具链接 `bootasm.o` 以及 `bootmain.c` 来生成 `bootblock.o` ，通过 `objcopy` 工具生成了 `bootblock` ，同时通过 `objdump` 工具生成 `bootblock.o` 的文件组成，并最后通过 `sign.pl` 脚本检查 `bootblock` 的大小是否超过限制；下面考察 `kernel` 的生成过程，首先它有 `$(OBJS)` , `entry.o` , `entryother` , `initcode` , `kernel.ld` 的依赖； `$(OBJS)` 是一系列目标文件合集的变量，由于这些目标文件没有明确各自的编译命令，因此它们的编译命令由通配符编

写的命令统一推测生成，`entry.o` 同理，`initcode` 与 `entryother` 分别由 `initcode.S`，`entryother.S` 通过 `gcc` 工具编译生成，再通过链接工具与 `objcopy` 工具生成，最后 `kernel.ld` 文件指导了 `kernel` 中各项依赖文件的依赖关系，由 `ld`，`objcopy` 工具生成 `kernel`，并且通过 `objdump` 工具生成了 `kernel` 可执行文件内部结构；最后考察 `fs.img` 的生成过程，`fs.img` 的生成过程与上述过程有些不同，他的所有依赖文件包括 `$(UPPROG)` 定义的一系列需要上传入系统磁盘的可执行文件，以及 `README` 文本文件，最终通过 `mkfs` 可执行文件执行外部代码生成 `fs.img`，其中 `$(UPPROG)` 定义的可执行文件通过 `Makefile` 中写好的通配符规则编译；至此，已经将 `xv6.img` 的生成部分分析结束，但是 `Makefile` 中还有其他的一系列命令，包括生成 `xv6memfs.img`（将 `xv6` 磁盘部分保存在内存中的版本），`clean` 命令，自动生成文档的 `print` 命令，调用虚拟机运行系统的 `bochs`，`qemu` 等等其他命令，在此不再赘述。

2. 问题：xv6如何做准备（建立GDT表等）并进入保护模式的?

   答：`xv6` 所做的准备包括:

   1. 禁用所有中断;
   2. 将 `%ds`，`%es`，`%ss` 段处理器置零;
   3. 开启 `A20 port(0x64,0x60)` 启用第21位地址;
   4. 将 `gdt` 指针指向预先建立好的GDT表;
   5. 使用 `CR0_PE` 初始化 `%cr0` 寄存器，启用保护模式;
   6. 使用 `ljmp` 命令重置 `%cs`，`%eip` 寄存器，使之指向32位代码段，并正式进入保护模式。

3. 问题：引导程序如何读取磁盘扇区的？又是如何加载ELF格式的OS的?

   答：为了回答第一个问题，考察 `readsect(void *dst, uint offset)` 函数，该函数描述了从IDE磁盘中读取一个 `sector` 的过程，首先需要等待磁盘准备好接收下一条指令，调用 `waitdisk()` 函数，循环查看 `0x1F7` 端口信号；等待结束后，通过 `outb` 指令向一系列磁盘端口写入参数，最后给 `0x1F7` 端口写入 `0x20` 指令，让磁盘开始工作，按照传入的参数独处数据到缓存区等到被取出；再次调用 `waitdisk()` 函数，等待磁盘读完数据；等待结束后，从 `0x1F0` 端口取出数据；这就是从磁盘扇区读取数据的完整过程。为了回答第二个问题，考察 `boomain(void)` 函数，该函数描述了引导程序将磁盘中的内核代码加载到内存中的过程；首先，直接读取磁盘中开头的4096个字节的数据到 `elf` 结构体中，用于检查 `elf` 结构体中的 `magic` 成员是否等于 `ELF_MAGIC`，也即检查磁盘中文件是否是一个合法的ELF可执行文件；若合法，则开始以 `segment` 为单位逐个读入OS，每个 `segment` 内部又有若干 `section`，调用 `readsect` 函数即可，至此，`bootmain` 已经将内核代码加载入内存，通过 `elf` 结构体中的 `entry` 成员获得内核代码的入口地址，调用 `entry()` 就将控制权转交给内核了。

阅后心得

1. 仅仅是一个玩具操作系统的引导代码就如此复杂，在现实情况中成熟的操作系统的引导会更加复杂吧。就像文档中最后说道的，成熟的操作系统引导不仅仅可以从磁盘中引导，还应当可以从光驱，软盘，甚至网络设备中引导，因此真正的引导程序本身就是一个小的操作系统。
2. 汇编代码给我阅读过程中造成了很多的困扰，虽然之前简单的接触过一种简单的玩具汇编语言，但是和

真正的x86汇编比起来，还是相差太多了，为了更好的理解操作系统，学习好汇编是必不可少的。
3. 看过Makefile的写法之后可以说是非常震撼，虽然之前写的代码都是通过自己手写Makefile编译的，但是操作系统的Makefile的编写涉及到了很多技巧，让我学到了很多。

一些疑惑之处

1. 从硬盘读取数据的代码中，有五行都是通过 `outb` 向磁盘端口传参，但是我并不是非常理解这些参数的意义。
2. 32位保护模式下的寻址模式感觉文档并没有讲的非常详细，因此只能在网上查阅资料了解GDT表以及段寄存器相关知识，希望可以在文档中加上这一部分的讲解，这对于为什么在进入保护模式后需要对段寄存器重新赋值这一问题的理解有帮助。