练习 4：分析 bootloader 加载 ELF 格式的 OS 的过程（要求在报告中写出分析）。

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。

（1）bootloader 是如何读取硬盘扇区的？

（2）bootloader 是如何加载 ELF 格式的 OS 的？

提示：可阅读 2.3.2 节中的“硬盘访问概述”和“ELF 执行文件格式概述”。

答：

（1）阅读 bootmain.c 中的代码可以知道，bootloader 首先通过 `readseg` 函数读取硬盘扇区，`readseg` 函数内部调用了 `readsect` 函数来每次读出一个扇区，`readsect` 函数中首先调用了 `waitdisk` 函数，`waitdisk` 函数的代码如下：

```
36  /* waitdisk - 等待磁盘准备好 */
37  static void
38  waitdisk(void) {
39      while ((inb(0x1F7) & 0xC0) != 0x40)
40          /* 什么都不做 */;
41  }
```

表 1 命令寄存器组

I/O 地址	读(主机从硬盘读数据)	写(主机数据写入硬盘)
0x1F0	数据寄存器	数据寄存器
0x1F1	错误寄存器(只读寄存器)	特征寄存器
0x1F2	扇区计数寄存器	扇区计数寄存器
0x1F3	扇区号寄存器或 LBA 块地址 0~7	扇区号或 LBA 块地址 0~7
0x1F4	磁道数低 8 位或 LBA 块地址 8~15	磁道数低 8 位或 LBA 块地址 8~15
0x1F5	磁道数高 8 位或 LBA 块地址 16~23	磁道数高 8 位或 LBA 块地址 16~23
0x1F6	驱动器/磁头或 LBA 块地址 24~27	驱动器/磁头或 LBA 块地址 24~27
0x1F7	状态寄存器	命令寄存器

表 2 IDE 状态寄存器

位	意义
0	ERR，错误(ERROR)，该位为 1 表示在结束前次的命令执行时发生了无法恢复的 错误。在错误寄存器中保存了更多的错误信息。
1	IDX，反映从驱动器读入的索引信号。
2	CORR，该位为 1 时，表示已按 ECC 算法校正硬盘的读数据。
3	DRQ，为 1 表示请求主机进行数据传输(读或写)。
4	DSC，为 1 表示磁头完成寻道操作，已停留在该道上。
5	DF，为 1 时，表示驱动器发生写故障。
6	DRDY，为 1 时表示驱动器准备好，可以接受命令。
7	BSY，为 1 时表示驱动器忙(BSY)，正在执行命令。在发送命令前先判断该位。

一般主板有 2 个 IDE 通道，每个通道可以接 2 个 IDE 硬盘。访问第一个硬盘的扇区是通过设置 I/O 地址寄存器 0x1F0~0x1F7 实现的，磁盘 I/O 地址读、写时对应的命令寄存器组

(Task File Registers)如表 1 所示。waitdisk 函数通过不断的从 0x1F7 读取磁盘的状态，并判断返回值的最高两位，直到最高两位（第 7 位、第 6 位）为 01，0x1F7 的每一位的含义如表 2 所示。[1]

readsect 函数的基本功能是读取一个硬盘扇区，其中 dst 是存储数据的位置，secno 是扇区的编号，outb 中相关端口的作用见表 1，代码如下：

```
43  /* readsect - 读取 secno 扇区将数据存入 dst */
44  static void
45  readsect(void *dst, uint32_t secno) {
46      // 等待磁盘准备好
47      waitdisk();
48
        // 第二个参数是读取扇区的个数
49      outb(0x1F2, 1);
        // 输入 LBA 参数的 0-7 位
50      outb(0x1F3, secno & 0xFF);
        // 输入 LBA 参数的 8-15 位
51      outb(0x1F4, (secno >> 8) & 0xFF);
        // 输入 LBA 参数的 16-23 位
52      outb(0x1F5, (secno >> 16) & 0xFF);
        // 输入 LBA 参数的第 24-27 位，第四位为 0 表示从主盘读取，其余位被强制置为 1
53      outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
        // 0x20 命令 - 读扇区
54      outb(0x1F7, 0x20);
55
56      // 等待磁盘准备好
57      waitdisk();
58
59      // 读一个扇区
60      insl(0x1F0, dst, SECTSIZE / 4);
61  }
```

其中 insl 在 libs/x86.h 中通过内联汇编实现，其作用是从端口 port 读取 cnt 个数据块到 addr 指向的内存区域中，每个数据块的大小是 4 个字节(uint32_t 是 4 个字节，uint32_t 在 libs/defs.h 的第 21 行中被宏定义为 unsigned int)，代码如下：

```
45  Static inline void
46  insl(uint32_t port, void *addr, int cnt) {
```

[1] 徐小玲. IDE 接口硬盘读写技术[J]. 电子科技大学学报, 2002, 31(6): 637-638.

```
47      asm volatile (
48              "cld;"
49              "repne; insl;"
50              : "=D" (addr), "=c" (cnt)
51              : "d" (port), "0" (addr), "1" (cnt)
52              : "memory", "cc");
53  }
```

在 boot/bootmain.c 中，SECTSIZE 在第 33 行被宏定义为 512，即一个扇区的大小是 512 字节，第 60 行代码中除以 4 是因为数据块的大小为 4 个字节。

readseg 函数通过调用 readsect 函数，每次读取一个扇区(512 字节)，从 offset 所在扇区(注意 offset 是相对 1 扇区的偏移量，第 0 扇区是 bootblock 引导区)开始，复制到 va + count 所在扇区，实际复制的字节数通常会超过 count 值，代码如下：

```
63  /* *
64   * readseg - 从相对内核起始位置(1 扇区)的 offset 处读 count
    个字节到虚拟地址 va 中。
65   * 复制的内容可能比 count 个字节多。
66   * */
67  static void
68  readseg(uintptr_t va, uint32_t count, uint32_t offset)
    {
        // uintptr_t 在 libs/defs.h 第 31 行被定义为 uint32_t
69      uintptr_t end_va = va + count;
70
71      // 向下舍入到扇区边界
72      va -= offset % SECTSIZE;
73
74      // 从字节转换到扇区；kernel 开始于扇区 1
75      uint32_t secno = (offset / SECTSIZE) + 1;
76
77      // 如果这个函数太慢，则可以同时读多个扇区。
78      // 我们写的字节数会超过 count，但这并不重要：
79      // 因为是以内存递增的次序加载的。
80      for (; va < end_va; va += SECTSIZE, secno ++) {
81          readsect((void *)va, secno);
82      }
83  }
```

（2）ELF 文件格式是类 UNIX 操作系统上二进制文件的标准格式，它由 ELF 头(ELF header)、程序头表(Program header table)、节(Section)和节头表(Section header table)构成[2]，其构成图如图 1 所示：

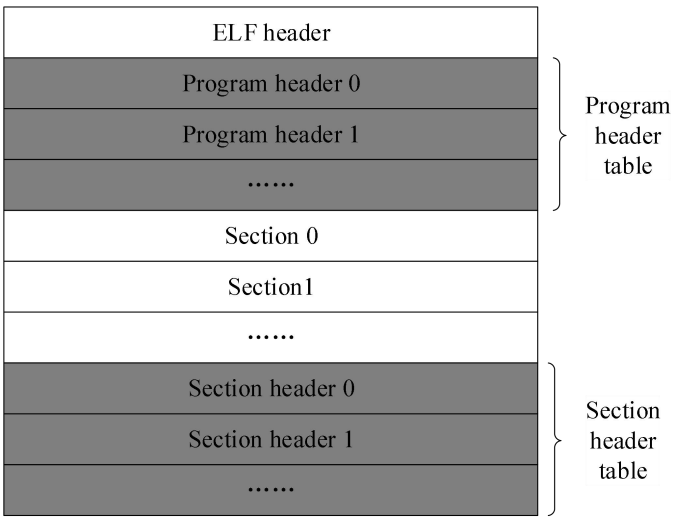

图 1 ELF 文件格式的布局图

ELF 头在文件开始处描述了整个文件的控制结构，它定义在 libs/elf.h 中，定义如下：

```
8   /* 文件头 */
9   struct elfhdr {
10      uint32_t e_magic;     // 必须等于 ELF_MAGIC
11      uint8_t e_elf[12];
12      uint16_t e_type;      // 1=relocatable,
    2=executable, 3=shared object, 4=core image
13      uint16_t e_machine;   // 3=x86, 4=68K, etc.
14      uint32_t e_version;   // file version, always 1
15      uint32_t e_entry;     // 程序入口的虚拟地址
16      uint32_t e_phoff;     // program header 表的位置偏移
    或 0
17      uint32_t e_shoff;     // file position of section
    header or 0
18      uint32_t e_flags;     // architecture-specific
    flags, usually 0
19      uint16_t e_ehsize;    // size of this elf header
20      uint16_t e_phentsize; // size of an entry in program
    header
21      uint16_t e_phnum;     // program header 表中的入口数
    目或 0
```

[2] 百度百科. https://baike.baidu.com/item/ELF/7120560?fr=aladdin.

```
22      uint16_t e_shentsize; // size of an entry in section
    header
23      uint16_t e_shnum;     // number of entries in section
    header or 0
24      uint16_t e_shstrndx;  // section number that
    contains section name strings
25  };
```

我们可以使用 readelf -h 命令来查看一个目标文件的 elf 头，如图 2 所示：

```
chy@chy-VirtualBox:~$ readelf -h 1.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          248 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         11
  Section header string table index: 8
```

图 2 使用 readelf -h 查看目标文件的 elf 头

首先，第一个 magic，叫做魔数，这个主要是程序用来确认读入的是否是 elf 文件头，其中，第一个 7f 是默认的，后面的 45、4c、46 就是 E、L、F 三个大写字母的 ASCII 码值，后面的 01 没有实际意义。每次程序在读取 elf 头文件的时候，都会确认魔数是否正确，以防读入的不是 elf 文件。[3]

Program header 描述了一个段或者系统准备程序执行所必需的其他信息，定义如下：

```
27  /* 程序头 */
28  struct proghdr {
29      uint32_t p_type;   // 段类型
30      uint32_t p_offset; // 段相对文件头的偏移值
31      uint32_t p_va;     // 段的第一个字节将被放到内存中的虚拟
    地址
```

[3] fang92. C 语言的 ELF 文件格式学习[OL]. http://www.cnblogs.com/fang92/p/4782730.html, 2015-08-30.

```
32      uint32_t p_pa;     // physical address, not used
33      uint32_t p_filesz; // size of segment in file
34      uint32_t p_memsz;  // 段在内存映像中占用的字节数
35      uint32_t p_flags;  // read/write/execute bits
36      uint32_t p_align;  // required alignment,
    invariably hardware page size
37  };
```

在 bootmain 中，先调用 `readseg` 函数从 ucore 内核镜像偏移为 0 处读入一页(8 个扇区)到内存 0x10000 处，ELFHDR 在 boot\bootmain.c 的第 34 行中被定义为了指向 0x10000 处 elfhdr 结构的指针，因此这行代码就是为了把这一页作为 elf 文件头读入。

使用 qemu 调试在 bootmain 函数开始处设置断点，然后执行第一条语句后查看 0x10000 处的值如图 3 和图 4 所示：

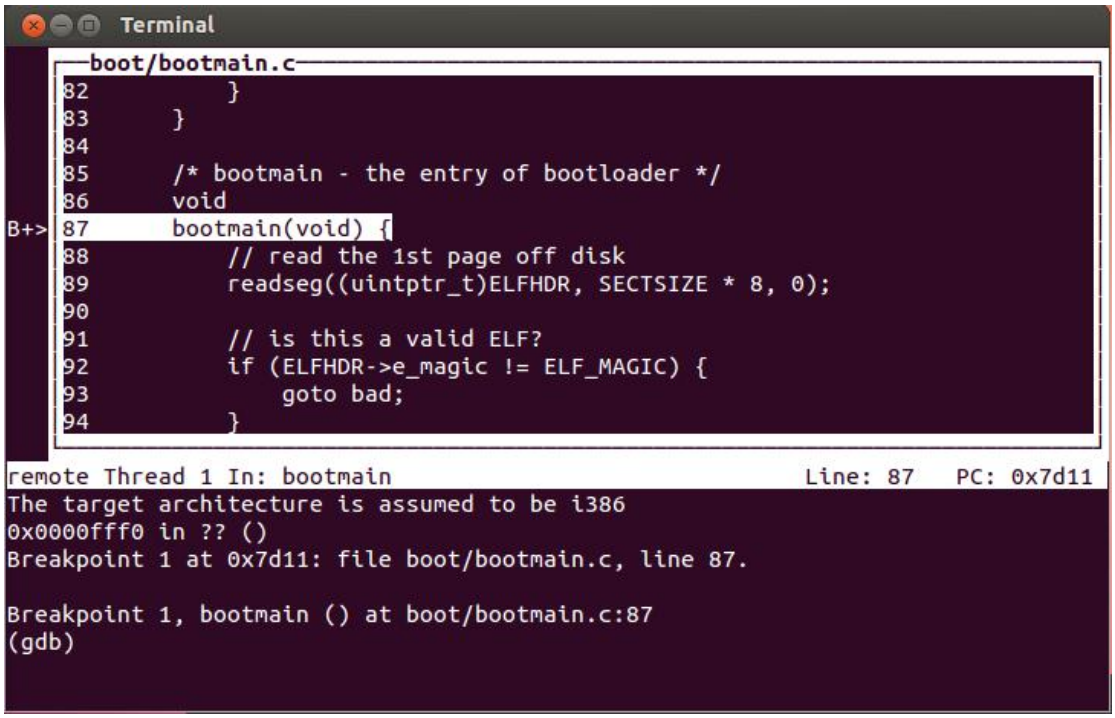

图 3 在 bootmain 处设置断点

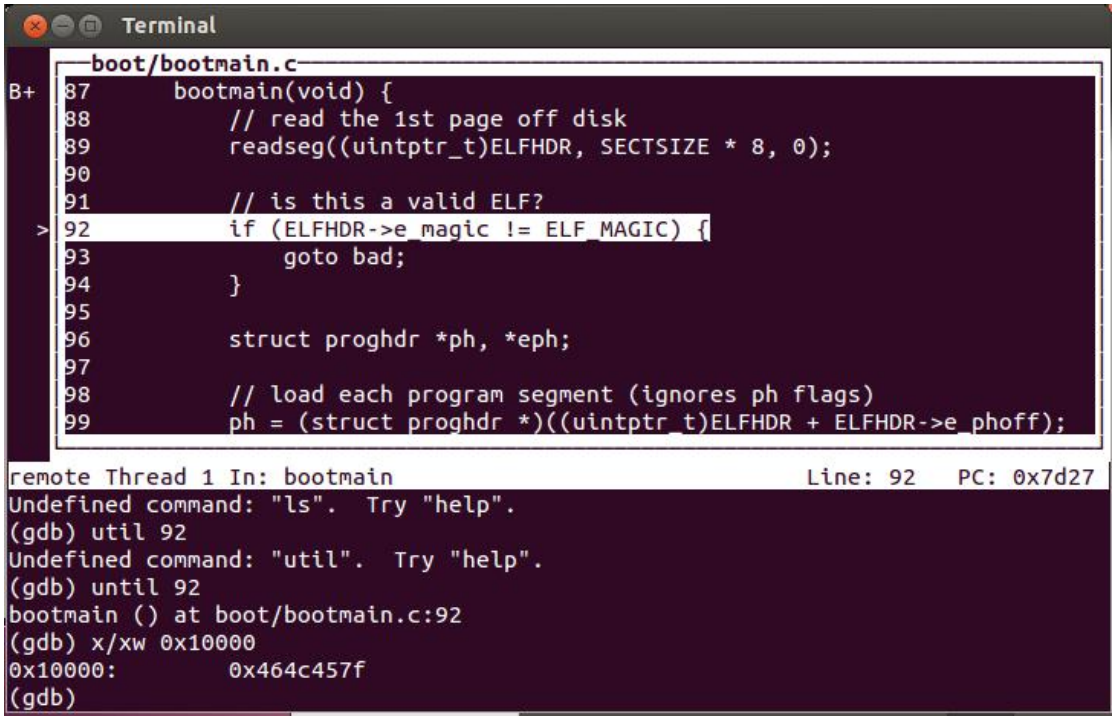

图 4 执行第一条语句后查看 0x10000 处的内容

从这里也可以看出，读出的内容确实是 0x464C457F，与 libs\elf.h 中第 6 行的 ELF_MAGIC 的宏定义一致，因此执行第 92 行代码的作用便是判断是否为合法的 elf 头，若不合法，跳到 bad 处继续执行并进入死循环。若合法，则将继续执行，第 96-103 行代码如下：

```
 96      struct proghdr *ph, *eph;
 97
 98      // 加载每个程序段（忽略 ph 标志）
 99      ph = (struct proghdr *)((uintptr_t)ELFHDR +
     ELFHDR->e_phoff);
100      eph = ph + ELFHDR->e_phnum;
101      for (; ph < eph; ph ++) {
102          readseg(ph->p_va & 0xFFFFFF, ph->p_memsz,
     ph->p_offset);
103      }
104
105      // 从 ELF 头中调用入口点
106      // 注意：不会返回
107      ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

elf 文件头有描述 Program header 应加载到内存什么位置的描述表，第 99 行读取出来将之存入 ph，第 100 行在 ph 上加上了 Program header 中的项数并存入 eph，后面的循环遍历

了 Program header 的每一项，并按照 Program header 的描述，将 elf 文件中的数据载入内存。最后，执行第 107 行代码，从 ELF 头中调用入口点。[4][5]

[4] Bendawang. 操作系统 ucore lab1 实验报告[OL]. https://blog.csdn.net/qq_19876131/article/details/51706973, 2016-06-18.
[5] 张慕晖. 操作系统实验（1）：系统软件启动过程[OL]. http://ilovestudy.wikidot.com/operating-system-lab-1#toc21, 2018-03-02.