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

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]](#footnote-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  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | /\* readsect - 读取secno扇区将数据存入dst \*/  static void  readsect(void \*dst, uint32\_t secno) {  // 等待磁盘准备好  waitdisk();  // 第二个参数是读取扇区的个数  outb(0x1F2, 1);  // 输入LBA参数的0-7位  outb(0x1F3, secno & 0xFF);  // 输入LBA参数的8-15位  outb(0x1F4, (secno >> 8) & 0xFF);  // 输入LBA参数的16-23位  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  outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);  // 0x20命令 - 读扇区  outb(0x1F7, 0x20);  // 等待磁盘准备好  waitdisk();  // 读一个扇区  insl(0x1F0, dst, SECTSIZE / 4);  } |

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  46  47  48  49  50  51  52  53 | Static inline void  insl(uint32\_t port, void \*addr, int cnt) {  asm volatile (  "cld;"  "repne; insl;"  : "=D" (addr), "=c" (cnt)  : "d" (port), "0" (addr), "1" (cnt)  : "memory", "cc");  } |

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  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | /\* \*  \* 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。  \* 复制的内容可能比count个字节多。  \* \*/  static void  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  uintptr\_t end\_va = va + count;  // 向下舍入到扇区边界  va -= offset % SECTSIZE;  // 从字节转换到扇区；kernel开始于扇区1  uint32\_t secno = (offset / SECTSIZE) + 1;  // 如果这个函数太慢，则可以同时读多个扇区。  // 我们写的字节数会超过count，但这并不重要：  // 因为是以内存递增的次序加载的。  for (; va < end\_va; va += SECTSIZE, secno ++) {  readsect((void \*)va, secno);  }  } |

（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]](#footnote-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  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /\* 文件头 \*/  struct elfhdr {  uint32\_t e\_magic; // 必须等于ELF\_MAGIC  uint8\_t e\_elf[12];  uint16\_t e\_type; // 1=relocatable, 2=executable, 3=shared object, 4=core image  uint16\_t e\_machine; // 3=x86, 4=68K, etc.  uint32\_t e\_version; // file version, always 1  uint32\_t e\_entry; // 程序入口的虚拟地址  uint32\_t e\_phoff; // program header表的位置偏移或0  uint32\_t e\_shoff; // file position of section header or 0  uint32\_t e\_flags; // architecture-specific flags, usually 0  uint16\_t e\_ehsize; // size of this elf header  uint16\_t e\_phentsize; // size of an entry in program header  uint16\_t e\_phnum; // program header表中的入口数目或0  uint16\_t e\_shentsize; // size of an entry in section header  uint16\_t e\_shnum; // number of entries in section header or 0  uint16\_t e\_shstrndx; // section number that contains section name strings  }; |

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：

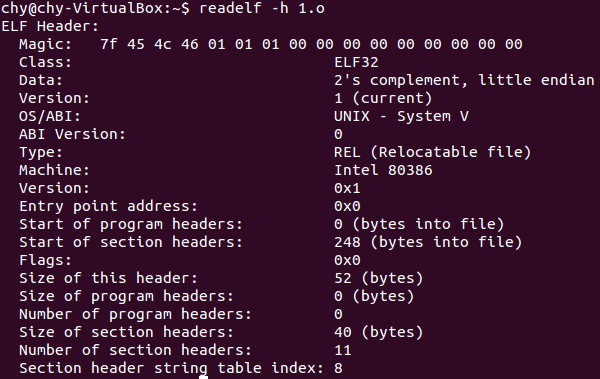


图2 使用readelf -h查看目标文件的 elf头

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]](#footnote-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  28  29  30  31  32  33  34  35  36  37 | /\* 程序头 \*/  struct proghdr {  uint32\_t p\_type; // 段类型  uint32\_t p\_offset; // 段相对文件头的偏移值  uint32\_t p\_va; // 段的第一个字节将被放到内存中的虚拟地址  uint32\_t p\_pa; // physical address, not used  uint32\_t p\_filesz; // size of segment in file  uint32\_t p\_memsz; // 段在内存映像中占用的字节数  uint32\_t p\_flags; // read/write/execute bits  uint32\_t p\_align; // required alignment, invariably hardware page size  }; |

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：

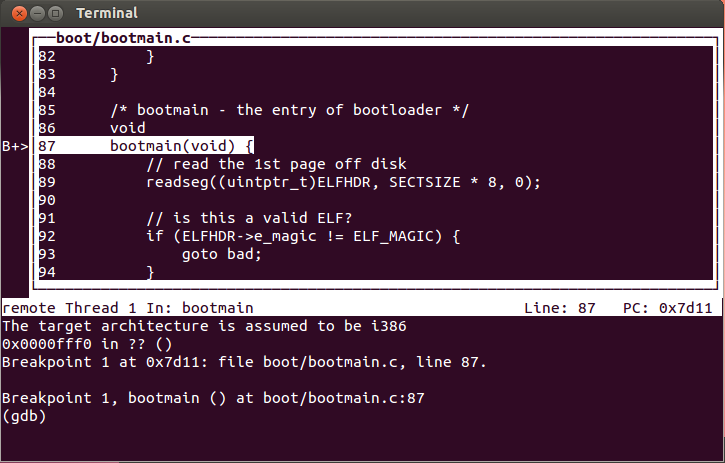


图3 在bootmain处设置断点

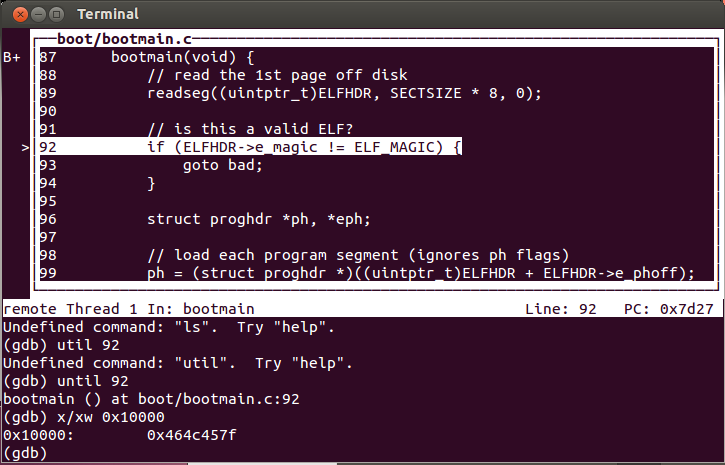


图4 执行第一条语句后查看0x10000处的内容

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  97  98  99  100  101  102  103  104  105  106  107 | struct proghdr \*ph, \*eph;  // 加载每个程序段 (忽略ph标志)  ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);  eph = ph + ELFHDR->e\_phnum;  for (; ph < eph; ph ++) {  readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);  }  // 从ELF头中调用入口点  // 注意: 不会返回  ((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]](#footnote-4)][[[5]](#footnote-5)]

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. [↑](#footnote-ref-1)
2. [] 百度百科. https://baike.baidu.com/item/ELF/7120560?fr=aladdin. [↑](#footnote-ref-2)
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. [↑](#footnote-ref-3)
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. [↑](#footnote-ref-4)
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. [↑](#footnote-ref-5)