

# Project1 Bootloader 设计文档

中国科学院大学

[姓名] 袁峰

[学号] 2015K8009929008

[日期] 2017.9.25

## 1. Bootblock 设计流程

### (1) Bootblock 主要完成的功能

bootblock 是在系统启动时首先执行的程序。

在任务二中, bootblock 的功能是直接输出一个字符串 “Welcome to OS”。根据任务中介绍的, PMON 下的字符输出为串口输出, 地址为 0xbfe48000, 输出字符串可以通过不断向该地址写字符实现。因此只需要写一个循环不断将字符存储到相应内存地址就可以完成输出。

在任务三中, bootblock 要完成的功能是将 kernel 的程序从硬盘加载到内存。kernel 程序首先由 kernel.c 经过编译生成 kernel 的 elf 文件, 然后在由 createimage 程序将 bootblock 和 kernel 一起制作成镜像, 并拷到 sd 卡上。系统启动时, 会自动将 sd 卡上的第一个扇区的内容(即 bootblock)复制到内存 0xa0800000 处, 并从 0xa0800030 处开始执行第一条指令。但 sd 卡上第二个扇区的内容(即 kernel)不会被自动复制到内存, 也就没有办法执行。因此任务三中将 kernel 拷到内存的任务将有 bootblock 来完成。

### (2) Bootblock 被载入内存后的执行流程

bootblock 被载入内存后, 将自动从 0xa0800030 开始执行第一条指令。

### (3) Bootblock 如何调用 SD 卡读取函数

根据任务中介绍, PMON 中的读盘函数的地址为 0x8007b1a8, 因为只需在 bootblock 中跳转到该地址就可以。由于 MIPS 是通过寄存器传参, 根据 MIPS 指令规范, 传参寄存器为 \$4, \$5, \$6 和 \$7。读盘函数需要传递三个参数, 第一个是读取的目的地址, 即读取的数据在内存存放的位置, 由于 kernel 将要被放在 sd 中第二个扇区, 一个扇区大小为 512, 相应的对应到内存中也就是 0xa0800200, 而且在 Makefile 文件中也可以找到该地址。第二个参数为 SD 卡内部的偏移量, 从该处开始读取。同样的, 由于 kernel 被 createimage 文件制作在了 image 的第二个扇区, 因此偏移量为 0x200, 也就是 512。第三个参数为要读取的字节数, 这个数据可以通过反汇编生成的 kernel 文件来获取, 通过 mipsel-linux-objdump -D kernel 命令, 我们可以查到反汇编代码。

```

Disassembly of section .rodata.str1.4:
a08002a8 <.rodata.str1.4>:
a08002a8:      73277449      0x73277449
a08002ac:      72656b20      0x72656b20
a08002b0:      216c656e      addi    t4,t3,25966
a08002b4:      00000a0d      break   0x0,0x28
Disassembly of section .data:
a08002c0 <_fdata>:
...
Disassembly of section .ctors:
a0800300 <__CTOR_LIST__>:
...
Disassembly of section .dtors:
a0800308 <__DTOR_LIST__>:

```

红框处是该输出的最后一行，加上该语句本身，我们算出 kernel 的大小为 0x10b，因此读取的字节数必须大于该大小，当然，取个更大的值也可以。

#### (4) Bootblock 如何跳转至 kernel 入口

先要找到 kernel 中的 main 函数地址，同样通过反汇编来查看。

```

a0800260:      8fb00010      lw      s0,16(sp)
a0800264:      03e00008      jr      ra
a0800268:      27bd0020      addiu   sp,sp,32
a080026c <main>:
a080026c:      3c04a080      lui     a0,0xa080
a0800270:      27bdffe8      addiu   sp,sp,-24
a0800274:      afbf0010      sw      ra,16(sp)
a0800278:      0c200087      jal     a080021c <printstr>
a080027c:      248402a8      addiu   a0,a0,680
a0800280:      8fbf0010      lw      ra,16(sp)

```

因此 kernel 的入口地址即为 0xa080026c，只需在 bootblock 完成将 kernel 载入内存后再跳转到该地址，便可以接着运行 kernel 程序。

#### (5) 任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

在完成该部分时，首先遇到的问题是上学期汇编语言课程学习的是 X86 的汇编，对 MIPS 指令集的了解较少，因此在编写 bootblock.s 先查看了 MIPS 指令手册，大致了解了有哪些指令可以用来使用。

其次，在调用函数时，一开始使用的是 j 指令，在反复尝试发现不成功，我查看了相关资料，发现调用函数时应该使用 jal 指令，该指令会自动将 PC+8 的地址放置到 ra (\$31) 寄存器，然后在子程序中使用 jr \$31 便可以调回原程序。

## 2. Createimage 设计流程

(1) Bootblock 编译后的二进制文件、Kernel 编译后的二进制文件，以及 SD 卡 image 文件这三者之间的关系

bootblock 和 kernel 编译后的二进制文件是 elf 文件，其中包含了 elf 文件头、程序头表及各程序段，其中各程序段中有一些为可装载段，而这些段也就是需要通过 createimage 程序生成到 image 文件中的内容。

(2) 如何获得 Bootblock 和 Kernel 二进制文件中可执行代码的位置和大小

```

typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;

```

上图为 elf 文件头的结构，首先在 elf 文件头中我们可以通过 `e_phoff` 得知程序头表在文件中的偏移，从 `e_phentsize` 得知每一个程序头的大小，从 `e_phnum` 得知程序头的数量，因此我们可以先找到程序头表的开始位置，然后从第一个程序段开始检查。

```

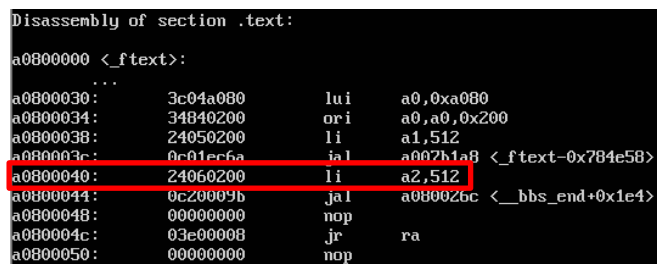
typedef struct {
    Elf32_Word       p_type;
    Elf32_Off        p_offset;
    Elf32_Addr       p_vaddr;
    Elf32_Addr       p_paddr;
    Elf32_Word       p_filesz;
    Elf32_Word       p_memsz;
    Elf32_Word       p_flags;
    Elf32_Word       p_align;
} Elf32_Phdr;

```

上图为程序头的结构，我们首先要判断 `p_type`，如果该数据为 `PT_LOAD`，那么就是我们真正要写入 image 文件的内容，也就是可执行代码。该程序段的位置可由 `p_offset` 得知，表示该程序段在文件中的偏移，通过 `p_filesz` 可以得出该程序段的大小。通过遍历所有程序头可以完成所有程序段的搜索。

### (3) 如何让 Bootblock 获取到 Kernel 二进制文件的大小，以便进行读取

我们在将 kernel 二进制文件中的程序段加载进 image 时，可以记录下加载的程序段的总长度。



```

Disassembly of section .text:
a0800000 <_ftext>:
...
a0800030: 3c04a080    lui    a0,0xa080
a0800034: 34840200    ori    a0,a0,0x200
a0800038: 24050200    li     a1,512
a080003c: 0c01ec6a    jal    a007b1a8 <_ftext-0x784e58>
a0800040: 24060200    li     a2,512
a0800044: 0c20009b    jal    a08002bc <__bbs_end+0x1e4>
a0800048: 00000000    nop
a080004c: 03e00008    jr     ra
a0800050: 00000000    nop

```

通过反汇编 bootblock 文件，我们得到如上信息，可以看到，在 `a0800040` 的地址写入了需要读取的字节数，也就是我们之前计算的程序段的总长度。从地址可以看出，字节数被写在了 bootblock 文件开始后的 `0x40` 偏移，也就是 image 文件中的 `0x40` 位置，由于是小尾端，因此只需要将 image 中 `0x40` 处开始两个字节的大小修改为程序段总长度即可。

### (4) 任何在设计、开发和调试 createimage 时遇到的问题和解决方法

在此次实验之前，对 ELF 文件格式不太了解，因此在完成此次实验时首先完整了解了 ELF 文件的格式。并学习了如果通过各种头文件来找到所需要的数据，并定位到 ELF 文件中的相应段。

在写 `createimage.c` 的过程中，指针操作较多，因此在编写时十分容易出错。而且在调试

的过程中由于只能通过上板查看结果来判断是否成功，因此调试的环节也不太方便，前前后后插拔读卡器和实验板有近二三十次。

在编写的过程中，也参考了上学期计算机组成原理实验中有一个实验所提供的代码，其功能也是加载 elf 文件，因此从中得到了不少的借鉴。

### 3. 关键函数功能

#### 1、bootblock.s

```

1  .text
2  .globl main
3  main:
4      # check the offset of main
5      nop
6      nop
7      nop
8      nop
9      nop
10     nop
11     nop
12     nop
13     nop
14     nop
15     nop
16     nop
17
18     #need add code
19     #read kernel
20     li $4,0xa0800200
21     li $5,0x200
22     li $6,0x0
23
24     jal 0x8007b1a8
25
26     jal 0xa080026c
27
28     jr $31

```

这个程序整体思路比较清晰，显示准备好了读盘函数所需的三个参数，然后调用读盘函数，此时 kernel 已经被加载到内存，于是跳转到相应的首地址执行 kernel 程序。

其中第 22 行参数为 0 是因为该参数需要在 createimage 程序中将其进行修改。

#### 2、createimage.c

```

92  int main(int argc, char *argv[]){
93
94      if (argc>1 && strcmp(argv[1],"--extended")==0)
95          extend=1;
96
97      FILE *image=fopen("image","wb");
98
99      FILE *boot=fopen(argv[extend+1],"rb");
100      write_bootblock(&image,&boot);
101      fclose(boot);
102
103
104      int i=0;
105      for (i=extend+2; i<argc; i++){
106          FILE *kernel=fopen(argv[i],"rb");
107          write_kernel(&image,&kernel,argv[i]);
108          fclose(kernel);
109      }
110      num_section*=512;
111      fseek(image,0x40,SEEK_SET);
112      fwrite(&num_section,sizeof(num_section),1,image);
113      fflush(image);
114      fclose(image);
115      return 0;
116  }

```

主程序主要是完成了—extended 选项的判断，并且分别调用 write\_bootblock() 和 write\_kernel()，并且在 111 行完成了对 bootblock 程序中需要读盘的字节数进行了修改。

```

10  uint8_t extend = 0;
11  uint16_t num_section=0;
12  uint32_t base;
13
14  void write_bootblock(FILE **image_file, FILE **boot_file)
15  {
16      FILE *image=*image_file;
17      FILE *boot=*boot_file;
18      uint8_t buf[1024];
19      Elf32_Ehdr *boot_ehdr;
20      Elf32_Phdr *boot_phdr;
21      fread(buf,1,1024,boot);
22      boot_ehdr=(void *)buf;
23      uint32_t magic=0x464c457f;
24      uint32_t *boot_magic=(uint32_t *)buf;
25      assert(*boot_magic==magic);
26
27      int i=0;
28      for (boot_phdr=(void *)buf+boot_ehdr->e_phoff; i<boot_ehdr->e_phnum; i++){
29          if (boot_phdr[i].p_type==PT_LOAD){
30              num_section++;
31              base=boot_phdr[i].p_vaddr;
32              fseek(boot,boot_phdr[i].p_offset,SEEK_SET);
33              uint8_t newbuf[512];
34              memset(newbuf,0,512);
35              fread(newbuf,1,boot_phdr[i].p_filesz,boot);
36              fwrite(newbuf,1,512,image);
37              fflush(image);
38
39              if (extend){
40                  printf("bootblock image info\n");
41                  printf("sectors: %d\n",num_section);
42                  printf("offset of segment in the file: 0x%x\n",boot_phdr[i].p_offset);
43                  printf("the image's virtual address of segment in memory: 0x%x\n",boot_phdr[i].p_vaddr);
44                  printf("the file image size of segment: 0x%x\n",boot_phdr[i].p_filesz);
45                  printf("the size of write to the OS image: 0x%x\n",boot_phdr[i].p_memsz);
46                  printf("padding up to 0x%x\n",512);
47              }
48          }
49      }
50  }
51

```

write\_bootblock 函数首先解析了 elf 头文件，判断幻数是否正确，然后找到程序段表，并逐段进行解析，若判断为可装载段，先找到该程序段在文件中的偏移并将文件指针指向该处，然后按照程序段大小读取相应长度的内容，并将其写入 image 文件内。且如果在调用时有—extended 选项，则输入相关信息。

```

53  void write_kernel(FILE **image_file, FILE **kernel_file,char *filename)
54  {
55      FILE *image=*image_file;
56      FILE *kernel=*kernel_file;
57      uint8_t buf[1024];
58      Elf32_Ehdr *kernel_ehdr;
59      Elf32_Phdr *kernel_phdr;
60      fread(buf,1,1024,kernel);
61      kernel_ehdr=(void *)buf;
62      uint32_t magic=0x464c457f;
63      uint32_t *kernel_magic=(uint32_t *)buf;
64      assert(*kernel_magic==magic);
65      int i=0;
66      num_section=0;
67      for (kernel_phdr=(void *)buf+kernel_ehdr->e_phoff; i<kernel_ehdr->e_phnum; i++){
68          if (kernel_phdr[i].p_type==PT_LOAD){
69              num_section++;
70              base=kernel_phdr[i].p_vaddr;
71              fseek(kernel,kernel_phdr[i].p_offset,SEEK_SET);
72              uint8_t newbuf[512];
73              memset(newbuf,0,512);
74              fread(newbuf,1,kernel_phdr[i].p_filesz,kernel);
75              fwrite(newbuf,1,512,image);
76              fflush(image);
77
78              if (extend){
79                  printf("%s image info\n",filename);
80                  printf("sectors: %d\n",num_section);
81                  printf("offset of segment in the file: 0x%x\n",kernel_phdr[i].p_offset);
82                  printf("the image's virtual address of segment in memory: 0x%x\n",kernel_phdr[i].p_vaddr);
83                  printf("the file image size of segment: 0x%x\n",kernel_phdr[i].p_filesz);
84                  printf("the size of write to the OS image: 0x%x\n",kernel_phdr[i].p_memsz);
85                  printf("padding up to 0x%x\n",512);
86              }
87          }
88      }
89  }
90

```

write\_kernel 函数的内容与 write\_bootblock 几乎一致。