Project1 Bootloader 设计文档

中国科学院大学
李云志 2015K8009929014
2017.9.25

1. Bootblock 设计流程

1) Bootblock 主要完成的功能

a) 为读盘函数准备参数，并且跳转到读盘函数入口地址

b) 跳转到已被加载进内存的 kernel 的入口地址

2) Bootblock 被载入内存后的执行流程

a) loadboot 加载完成后，bootblock 程序从入口地址 0xa0800028 处开始执行

b) 准备调用读盘函数所需的三个参数：目的地址，SD 卡内部偏移量，读取的字节数。

代码如下：

```
li $4, 0xa0800200//目的地址
li $5, 0x200//SD 卡内部偏移量
li $6, 0x150//读取的字节数
```

c) 跳转至内存地址 0x8007b1a8 处执行读盘函数

d) 跳转至 kernel 入口地址：0xa080026c

e) kernel 函数执行

3) Bootblock 如何调用 SD 卡读取函数

Bootblock 先将读盘函数所需的三个参数准备好后，跳转到地址 0x8007b1a8 调用读盘函数

4) Bootblock 如何跳转至 kernel 入口

通过指令：jal 0xa080026c 跳转到 kernel 入口地址

5) 设计、开发、调试 bootblock 时遇到的问题及解决办法

a) 三个参数值以及跳转入口地址的确定：关于读取的目的地址以及 kernel 的入口地址只需利用 objdump 观察 kernel.c 生成的可执行文件，即可确定。

```
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)
a0800284:       00001021        move    v0,zero
a0800288:       03e00008        jr      ra
a080028c:       27bd0018        addiu   sp,sp,24
a0800290:       a0030014        sb      v1,20(zero)
```

例如确定 main 函数入口地址：

关于需要读取的字节数以及 SD 卡内部的偏移量只需通过 vim 打开生成的 image 文件，即可确定，截图如下：

我们可以看到 0x00000200 处为 kernel 文件，只需将读取字节参数设置为大于其文件字节数的某一常量即可。

```
00000200: 0400 8010 0000 0000 ffff 8424 ffff 8054  ...........$...T
00000210: ffff 8424 0800 e003 0000 0000 e0ff bd27  ...$...........'
00000220: 1000 b0af e4bf 023c 2180 8000 1027 0424  .......<!....'.$
00000230: 1400 b1af 1800 bfaf 8000 200c 0080 5134  .......... ...Q4
00000240: 9400 2008 0000 0292 0100 1026 0000 0292  .. .......&....
00000250: fdff 4054 0000 22a2 1800 bf8f 1400 b18f  ..@T.."...,.....
00000260: 1000 b08f 0800 e003 2000 bd27 80a0 043c  ........ ..'...<
00000270: e8ff bd27 1000 bfaf 8700 200c a802 8424  ...'...... ....$
00000280: 1000 bf8f 2110 0000 0800 e003 1800 bd27  ....!..........'
00000290: 1400 03a0 0000 0000 0000 0000 0000 0000  ................
000002a0: 0000 0000 0000 0000 4974 2773 206b 6572  ........It's ker
000002b0: 6e65 6c21 0d0a 0000 0000 0000 0000 0000  nel!............
```

b) 关于跳转指令的选取：
一开始，我使用的指令为 j，但发现即使入口地址写对后，也无法跳转至 kernel 的 main 函数处，后来发现需要先将函数返回地址即 pc+8 存入 31 号寄存器，读盘函数返回后才会进入 kernelmain 函数的入口。

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

image 文件是通过 createimage 将 Bootblock 与 Kernel 编译后的可执行文件中需要 load 的部分提取后写入而组成的，如图：

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

a) 从 Elf header 中寻找程序头表的偏移 e_phoff

b) 创建循环变量，搜索每一个程序段中需要装载的部分，并获得相应的位置与大小（具体细节将在第三部分的代码中说明）

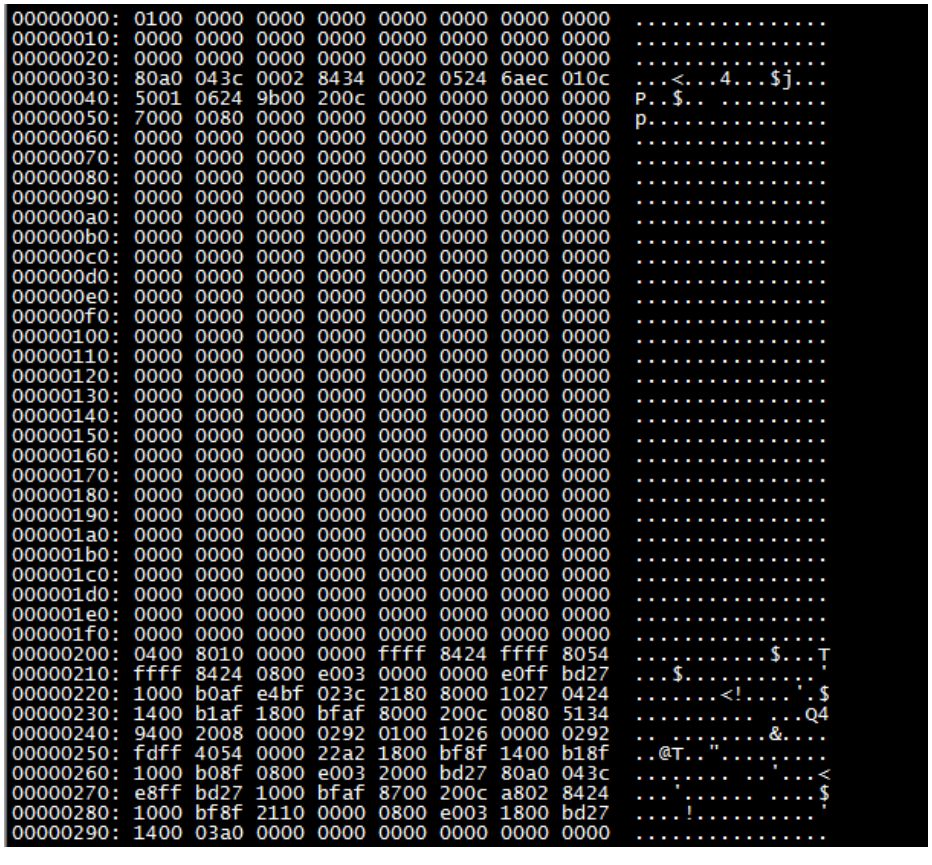
3) 如何让 Bootblock 获取到 Kernel 二进制文件的大小，以便进行读取

在写入 image 文件时，记录 kernel 所需的扇区数，并将此数值写入一个预设好的地址，之后即可通过对 bootblock.s 文件的编写来获取此数值，即 Bootblock 获取到了 Kernel 所要加载的文件的大小，即可进行读取。

代码如下：

```
fseek(image,0,SEEK_SET);
fwrite(&kernel_count,sizeof(kernel_count),1,image);
```

如图 0 地址被写入 1，为 kernel 所需扇区数



4) 设计、开发和调试 createimage 时遇到的问题和解决方法

a) 函数模块划分，createimage 分为四个部分：主函数，bootblock 写入，kernel 写入，打印信息。

且包含功能如下：

$$\text{main}\begin{cases} write\_bootblock\begin{cases} read\_exec\_file \\ extend\_opt \end{cases} \\ write\_kernel\begin{cases} read\_exec\_file \\ extend\_opt \\ count\_kernel\_sectors \end{cases} \\ record\_kernel\_sectors \end{cases}$$

PS: 函数名称及功能实现与任务书中有所差异，但功能全部完成

b) 开始对 Elf 文件了解并不深入，在研究 Elf 文件格式以及内容划分之后，继续完成实验内容，并对 Elf 文件头，组织结构，数据类型及数据结构有了较为深入的了解

c) 对 C 语言文件读写函数有些生疏，在写代码过程中，边学习，边实践。

3. 关键函数功能

1) 任务三的关键代码及功能已在前一部分说明的较为充分了，此处不再说明

2) 任务四：

a) bootblock.s 修改

关于动态加载 kernel 文件扇区数解决方法：createimage 文件将扇区数写入 bootblock 部分的空白区域：0x0 处，然后通过 bootblock.s 文件动态加载，代码如下：

```
li $4, 0xa0800200
li $5, 0x200
lw $6, main
sll $6, $6, 9

jal 0x8007b1a8
nop
jal 0xa080026c
```

b) bootblock 文件写入：

```
count_sector++;
uint32_t addr = ph[i].p_vaddr;
base=addr;
fseek(fp,ph[i].p_offset,SEEK_SET);
uint8_t newbuf[512];
memset(newbuf,0,512);
fread(newbuf,1,ph[i].p_filesz,fp);
fwrite(newbuf,1,512,image);
fflush(image);
```

由于 bootblock 文件不会超过 512，所以只需要开取 512 字节的数组读取文件并写入 image 即可，且扇区数加 1

c) 预先为 kernel 开取扇区：

```
uint8_t newbuf[512];
memset(newbuf,0,512);
if(offset+ph[i].p_memsz >(kernel_count+1)*512 )
{
    for(j= count_sector;j*512<offset+ph[i].p_memsz;j++)
    {
        fseek(fp,0,SEEK_END);
        fwrite(newbuf,1,512,image);
        fflush(image);
    }
}
```

完成的主要工作：判断开取扇区是否足够 -> 若不够则开取 512 字节扇区，全写入 0 -> 循环至扇区开取足够为止。

d) kernel 文件写入部分代码：

```
uint8_t *write_buf = malloc(ph[i].p_filesz);
fseek(fp, ph[i].p_offset, SEEK_SET);
fread(write_buf, 1, ph[i].p_filesz, fp);
fseek(image, offset, SEEK_SET);
fwrite(write_buf, 1, ph[i].p_filesz, image);
fflush(image);
free(write_buf);
```

e) 将 kernel 扇区个数写入 image 文件部分，独立存在于 main 函数中，代码如下：

```
fseek(image,0,SEEK_SET);
fwrite(&kernel_count,sizeof(kernel_count),1,image);
```

f) 输出 Elf 文件长度部分：

```
boot=fopen(argv[extend+1],"rb");
fseek(boot,0,2);
printf("length of bootblock = %d\n",ftell(boot));
fclose(boot);
for(i=extend+2;i<argc;i++)
{
    FILE *kernel=fopen(argv[i],"rb");
    fseek(kernel,0,2);
    printf("length of kernel = %d\n",ftell(kernel));
    fclose(kernel);
}
```

g) 判断是否需要输出信息部分代码：

其中 extend 为全局变量，由于子函数判断是否需要输出信息

```
if(strcmp(argv[1],"--extended")==0)
    extend=1;
```

■