Project1 Bootloader 设计文档

中国科学院大学
[王苑铮]
[2017.9.26]

1. Bootblock 设计流程

请至少说明以下内容

（1） Bootblock 主要完成的功能

把 kernel 从 sd 卡读取到内存中的指定位置，之后跳转到内核的起始地址让内核开始执行。

（2） Bootblock 被载入内存后的执行流程

依次把内核在内存中的起始地址、内核在 sd 卡中的偏移量、内核的大小存入函数参数寄存器，之后调用 pmon 的读盘函数将内核从 sd 卡读进内存，之后跳转到内核起始执行的函数地址

（3） Bootblock 如何调用 SD 卡读取函数

依次把内核在内存中的起始地址 addr、内核在 sd 卡中的偏移量 offset、内核的大小 size 存入函数参数寄存器，之后调用 pmon 的读盘函数，这个函数就会从 sd 卡偏移量 offset 开始，读出 offset 个字节，从 addr 起始存入内存

（4） Bootblock 如何跳转至 kernel 入口

人在算出 kernel 的入口地址后，通过 jal 指令跳转到这个地址

（5） 任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

① 第一次编写 bootblock 时，读盘函数的三个参数（内核在内存中的起始地址 addr、内核在 sd 卡中的偏移量 offset、内核的大小 size）全都不知道

解决方法：先随便填写三个数，然后用 make 编译，其打印结果中包含有我所需要的这三个参数，之后把这三个参数重新写进我的 bootblock.s 里，再重新 make 一遍，就得到有正确参数的 bootblock 了

② 把参数填进去后 bootblock 依然无法运行，怀疑是我有的参数填的不对

解决方法：offset 填的不对。我一开始填的是 kernel_phdr->poffset（因为打印信息里只有这个名字里有 offset），后来看 elf 的说明才知道这个是 kernel 起始地址在 kernel 的 elf 文件中的 offset，而不是 kernel 相对 sd 卡起始位置的 offset。读盘函数需要的 offset 是 padding up to 0x200 的 0x200。（一开始我没明白 pad up 在这里是什么意思，后来知道是填满扇区的意思，0x200 是十进制的 512，即一个扇区的大小）

③ 虚拟机显示已连接 sd 卡，并且用 lsblk 命令也能看到 sdb 被挂载了，并且也显示了 3.7G 的存储容量，但是 image 无法被 dd 进 sd 卡

解决方法：分别换用了新卡、新板子，都不成功，说明不是硬件问题。把程序发给小伙伴烧进他的 sd 卡，在板子上能成功运行，说明不是程序问题。后来用 fdisk –l /dev/sdb，却没有 sdb 的信息，这说明虽然 lsblk 显示 sd 卡挂载了，但实际上并没被挂载，所以才无法烧录。后来重启了虚拟机问题就解决了（只知其然，不知其所以然。怀疑是驱动或者虚拟机的问题。）

2. Createimage 设计流程

请至少说明以下内容

（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

组成关系：image 是由 bootblock 和 kernel 里的程序所组成的，bootblock 的程序在前面，并且填满一整个扇区（代码没填满的部分用 0）填上；kernel 在后面，并且也要把它占用的扇区填满。

调用关系：pmon 的启动程序读取 sd 卡中的 image，会先读第一扇区把 bootblock 放进内存，之后 bootblock 中的读盘函数再调用读盘函数把 kernel 读进内存（内存中也在 bootblock 后面的位置）并跳转到 kernel 的起始执行地址。

（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

位置：在 elf 文件中，先用 fread 从二进制文件中读出 ehdr 头，之后由 ehdr->e_phoff 找到第一个 Phdr 头，在由 phdr->p_offset 得到可执行代码的起始位置。

大小：ehdr->e_phnum 表示 elf 文件中 phdr 的数量，每个 phdr 都是一段可执行代码的头。将 ehdr->e_phnum 个 phdr 头都读出来，将它们的 phdr->p_filesz 相加，得到的就是可执行代码的大小

以上对 kernel 和 bootblock 都适用

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

在 task3 中，kernel 的 size 是已经写死的，这里 kernel 大小是可变的。我先用 objdump 反汇编，找到 load kernel 大小的语句所在的位置以及机器码，找到之后，在 createimage.c 中，在 record_kernel_sectors()里先通过 fseek 定位到那一条机器码的位置（图中的 0xa0800040）。该机器码的后四位是 kernel 的大小，我把一条后四位改为我的 kernel size 的机器码用 fwrite 写进 image，覆盖掉原本那条机器码就可以了。

```
bootblock:     file format elf32-tradlittlemips

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:       24060110        li      a2,272
a0800044:       0c20009b        jal     a080026c <__bbs_end+0x1dc>
a0800048:       00000000        nop
a080004c:       00000000        nop
a0800050:       80000070        lb      zero,112(zero)
```

还有另一种实现思路：bootblock 中，kernel 的 size 不是通过立即数写进寄存器，而是从一个特定的内存地址 lw 进寄存器。然后在 creatimage 中把 kernel 的 size 写进内存的那个位置。，这样 bootblocks 就可以把内存中的 kernel size load 进寄存器了。我是打算把那个数据存在 bootblock 的 main 下面第一个 nop 的位置。问了蒋老师，他说这次通过改立即数机器码（我采用的就是这种）就可以了，因为以后的实验中 main 下面可能不是 nop 指令，这种取巧的方法就不适用了。

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

① 一开始不知道文件指针是什么东西，以为文件就是内存中连续的一段，导致很多和文件相关的操作都错了（比如修改文件指针在文件中的位置，我直接用了类似于&image+ehdr->phoff 这种方式），导致完全运行不起来。
解决方式：后来知道文件是在外存里的，可以看出一种特殊的结构体，对文件进行操作需要用 c 语言的文件函数（fseek，fopen，fread，fwrite 等等）来完成

② 想用 fread 直接把 bootblock 里的东西读进 image，失败了
解决方案：fread 是把文件读进内存的，所以要先在内存中开出一个 buffer 出来，把文件用 fread 读进 buffer，再把 buffer 的内容用 fwrite 写进 image（对 kernel 同理）

③ Fread 没法把 Ehdr、Phdr 读给 ehdr、phdr
解决：需要先用 malloc 个 ehdr、phdr 两个指针分配空间

④ 以为 bootblock 的起始地址是 a0800030，所以算我要改的那条机器码（位于 a0800040）相对于 image 起始位置的偏移量我算成了 0x10

```
bootblock:     file format elf32-tradlittlemips

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:       24060110        li      a2,272
a0800044:       0c20009b        jal     a080026c <__bbs_end+0x1dc>
a0800048:       00000000        nop
a080004c:       00000000        nop
a0800050:       80000070        lb      zero,112(zero)
```

解决：其实 dumpfile 前面有一段没有打印出来，bootblock 的起始地址是 a0800000 而不是 a0800030，偏移量是 0x40

3. 关键函数功能

请列出你觉得重要的代码片段、函数或模块（可以是开发的重要功能，也可以是调试时遇到问题的片段/函数/模块）

我的代码贴图中有非常详细的注释。

Read_exec_file()函数：打开可执行文件，并读出它的 Ehdr 和它所有的 Phdr。

```
Elf32_Phdr * read_exec_file(FILE **execfile, char *filename, Elf32_Ehdr **ehdr)
{
    //为ehdr分配空间
    *ehdr = (Elf32_Ehdr*)malloc(sizeof(Elf32_Ehdr));
    if( ehdr== NULL ){
        printf("malloc %s ehdr error\n",filename);
        exit(0);
    }

    //打开可执行文件
    *execfile = fopen(filename,"rb"); //open elf file

    if(*execfile == NULL){
        puts("file error");
        return NULL;
    }

    //文件指针不是这么用的
//  *ehdr =(Elf32_Ehdr*) *execfile; //elf head is at the top of elf file,so elf head's addr is file's addr
//  phdr  =(Elf32_Phdr*) *execfile + (*ehdr)->e_phoff;

    //读出位于elf文件最前面的Ehdr
    fread(*ehdr,sizeof(Elf32_Ehdr),1,*execfile);

    //读Phdr
    //定位到第一个Phdr的位置
    fseek(*execfile,(*ehdr)->e_phoff,SEEK_SET);
    //给phdr分配空间
    //每个elf文件的ehdr只有一个，但是phdr可以有很多（ehdr->e_phnum个），这些Phdr都要读出来。
    //各个Phdr是挨在一起的，size也一样大，所以可以用数组的方式调用各个phdr
    Elf32_Phdr * phdr=(Elf32_Phdr*)malloc( ((*ehdr)->e_phnum) * sizeof(Elf32_Phdr));
    if( phdr== NULL ){
        printf("malloc %s phdr error\n",filename);
        exit(0);
    }
    fread(phdr,sizeof(Elf32_Phdr),(*ehdr)->e_phnum,*execfile);
    return phdr;
}
```

Write_kernel：先算出 kernel 需要的扇区数，在内存中开出相应大小的 buffer，把 kernel 的可执行代码读进 buffer，再把 buffer 内容写进 image

```
void write_kernel(FILE **image_file, FILE *kernel_file, Elf32_Ehdr *kernel_ehdr, Elf32_Phdr *kernel_phdr)
{
    //计算出kernel所需的size以及最终所需要填满的扇区数后，在内存中开出相应大小的buffer
    int size;
    int sector_num = count_kernel_sectors(kernel_ehdr,kernel_phdr,&size);
    char buf[sector_num*512];
    //把buffer先清为0，这样在把程序读进buffer后，再把buffer读进ssd后，扇区没用完的部分剩下的就是用0填满了
    memset(buf,'\0',sector_num*512);

    //分多次把各个phdr对应的可执行代码读进buffer，然后一次把buffer内容写进image
    int i,already_read;
    //挨个把每个phdr下面对应的可执行代码读进buffer
    for(i=0,already_read=0; i<kernel_ehdr -> e_phnum ; ++i){
        fseek(kernel_file,kernel_phdr[i].p_offset,SEEK_SET);
        fread(buf+already_read,kernel_phdr[i].p_filesz,1,kernel_file);
        already_read += kernel_phdr[i].p_filesz;
    }
    //把装有可执行代码的buffer内容写进image
    fwrite(buf,1,sector_num*512,*image_file);
    //image在外存里，而memset是清内存用的，这么用不对
    //memset(*image_file,0,kernel_phdr->p_filesz);
    return;
}
```

Record_kernel_sectors: //通过改机器码，改变 image 里 bootblock 的 kernel size 参数值

```
void record_kernel_sectors(FILE **image_file, Elf32_Ehdr *kernel_ehdr, Elf32_Phdr *kernel_phdr, int num_sec){
    //实现思路1:
    //这个imm初始化的值后四位是立即数，代表kernel的大小（目前是全0）。前面的位是操作码以及要load的寄存器号
    int imm[1]= {0x24060000};
    //后四位的立即数改为kernel的大小
    imm[0] += (num_sec*512);
    //定位到那条机器码所在的位置（它相对于文件起始位置的相对位置，是通过objdump找到的）
    fseek(*image_file,0x40,SEEK_SET);
    //写进新的机器码，覆盖原有机器码
    fwrite(imm,1,4,*image_file);

    /*//实现思路2:
    //注：我们用的都是冯诺依曼结构的计算机，数据和指令都用同一个ram，所以才能做到用数据覆盖原本是指令的内存空间
    //算出kernel大小
    int size[1];
    size[0] = num_sec*512;
    //定位到main的第一个nop地址（即image的开头地址）
    fseek(*image_file,0,SEEK_SET);
    //在内存中写入kernel的地址，覆盖掉第一个nop
    fwrite(imm,1,4,*image_file);
    */
    return;
}
```