Project1 Bootloader 设计文档

中国科学院大学
段江飞
2018 年 9 月 23 日

1. Bootblock 设计流程

（1）Bootblock 主要完成的功能

Bootblock 在任务一中通过调用 BIOS 函数 printstr 打印字符串“It’s a bootloader.”，在任务二中，Bootblock 将 kernel 加载到内存中，并跳转到 kernel 执行。

（2）Bootblock 被载入内存后的执行流程

Bootblock 在 sd 卡第一个扇区，被自动加载到内存中，由于链接脚本的原因，代码段放在了可执行文件的开头位置，所以 Bootblock 加载到内存中 0xa0800000 位置也是 Bootblock 执行的入口地址，从此地址开始依次执行 Bootblock 的代码，将要打印的字符串地址送到参数寄存器$a0，然后 jal 跳转到 printstr 执行打印字符串。

（3）Bootblock 如何调用 SD 卡读取函数

Bootblock 执行完 printstr 函数，会返回到调用函数的下一条指令（跳过分支延迟槽）继续执行，调用 read_sd_card 函数将 kernel 写入内存。SD 卡读取函数有三个参数，第一个参数是要移动数据到内存中的位置，对 kernel 而言，这个位置就是 0xa0800200，第二个参数是要移动数据在 SD 卡中的偏移量，Bootblock 占了第一个扇区，kernel 在第二个扇区，偏移量为 0x200，第三参数为要读取数据的大小，即 kernel 的大小，根据在 0xa08001ff 位置的 kernel 的扇区个数计算大小。依次将这三个参数传给参数寄存器$a0、$a1 和$a2，然后用 jal 跳转到 SD 卡读取函数地址开始执行，将 SD 卡上 kernel 读入内存。

（4）Bootblock 如何跳转至 kernel 入口

Bootblock 调用 SD 卡读取函数将 kernel 写入内存 0xa0800200 开始的区域，由于 ld.script 脚本将代码段放在了开头，因此 kernel 在内存中入口就是写入的位置，Bootblock 利用函数调用，用 jal 跳转到 0xa0800200 位置执行 kernel。

（5）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

A．地址与标号问题

调用函数的时候，直接利用文件里的标号，jal printstr，但是执行的时候会出现问题，仔细思考之后发现，这样的跳转跳转到了标号所在的位置执行，而不是调用后标号的值对应的地址执行函数，换成具体地址后解决了问题。

B．C 语言中 BIOS 函数调用

在 kernel 中调用打印字符串函数，也没有任何信息就不知道怎么调用，后来，在询问同学之后，发现函数指针的用法。

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，以及写入 SD 卡的 image 文件这三者之间的关系

写入 SD 卡的 image 文件由两部分组成，第一部分来自 Bootblock 编译后的二进制文件，是去掉了 ELF 头和文件头的代码段和数据等，在 SD 卡中的第一扇区，第二部分来自 Kernel 编译后的二进制文件，同样是去掉了 ELF 头和文件头的代码段和数据等，在 SD 卡中的第二扇区。

（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

Bootblock 和 Kernel 二进制文件开头都是 ELF 头和程序头，ELF 头结构体中的 e_phoff 变量表示程序头里 ELF 头的偏移量，可以找到程序头的位置，程序头结构体中的 p_offset 和 p_filesz 分别表示可执行代码段的偏移量和大小。

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

由于 Kernel 是内核，二进制文件大小可能很大，为了方便读取可以根据 kernel 的可执行代码段的文件大小计算出它所占的扇区个数，然后一个扇区一个扇区的进行读取写入。

（4）extended flag 打印

利用命令行参数识别 extended flag。

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

A．文件读取问题

打开文件读取写入的时候出错，因为 creatimage 要读取的是二进制文件，用 fopen 打开文件的时候，参数要设置为 rb，wb。

B．读文件特定位置和大小的内容

我只知道 fread 去读一定大小的文件，但是因为要找到程序头、代码段，读指针就要偏移，就去上网查取资料，找到 fssek 函数。

C．程序头

读可执行文件函数，要找到程序头的位置，我从文件中读取之后，拷贝到我分配的一块内存上，但是利用打印发现我根本就没有读到，百思不得其解，后来发现我读取是只读取了 ELF 头大小的文件，拷贝的时候，并没有将程序头读出来，所以出错，因此我修改了代码，一次读一个扇区，才解决了问题。

3. 关键函数功能

1. 函数指针

```
void __attribute__((section(".entry_function"))) _start(void)
{
    // Call PMON BIOS printstr to print message "Hello OS!"
    char str[] = "Hello OS!";
    printstr = (void *)0x8007b980;
    (*printstr)(str);
    return;
}
```

2. 函数调用

```
la $a0, msg
jal 0x8007b980
```

3. 读程序头

```
Elf32_Phdr *read_exec_file(FILE *opfile)
{
    Elf32_Ehdr *elf;
    Elf32_Phdr *Phdr = (Elf32_Phdr *)calloc(1, sizeof(Elf32_Phdr));
    uint8_t buf[512];

    fread(buf, 1, 512, opfile);
    elf = (Elf32_Ehdr *)buf;
    memcpy(Phdr, (void *)elf + elf->e_phoff, 32);
    return Phdr;
}
```

4. Bonus

（1）覆盖 bootblock 写入

将写入内存地址改为 Bootblock 起始地址，由于覆盖之后，从 SD 卡读取函数返回时，返回地址需要变为 0xa0800000，因此，手动写入 ra，跳到 SD 卡读取函数执行。

```
la $a0, 0xa0800200
li $a1, 0x200   #offset in sd card <- 0x200
li $a2, 0x200
lb $t0, 0xa08001ff
mul $a2, $t0, $a2
jal 0x8007b1cc  #read_sd_card
```

（2）修改 kernel

```
void (*printstr)(char *str);

void __attribute__((section(".entry_function"))) _start(void)
{
    // Call PMON BIOS printstr to print message "Hello OS!"
    char *str = "Hello OS!";
    printstr = (void *)0x8007b980;
    (*printstr)(str-0x200);
    asm("b -0x204");
    return;
}
```

在 kernel 里添加一个死循环，避免循环打印。调用 printstr 打印“Hello OS!”，但是我一直打印乱码，这个 bug 我找了一天，才发现编译的时候，str 指向只读数据区，调用打印字符串函数时，传入的参数是 str 的地址，由于 kernel 覆盖了 bootblock 的位置，整体上移了 0x200，导致按照原来地址找不到正确的数据，出现错误。所以对字符地址作了修改，当然，死循环那一段也是这样。