Project1 Bootloader 设计文档

中国科学院大学
陈灿宇
2018.9.26

1 Bootblock 设计流程

(1) Bootblock 主要完成的功能

Bootblock 负责机器启动时的引导过程, 机器将 Bootblock 加载进内存指定位置后开始执行其中的代码, 一般用于加载 Kernel 等。

(2) Bootblock 被载入内存后的执行流程

机器从 Bootblock 的 0x30 处开始执行, 首先将读取 SD 卡函数 read_sd_card() 所需的 3 个参数分别传进 4、5、6 号寄存器 (根据 MIPS 调用约定) , 然后使用 jal 将 PC+8 存入 31 号寄存器, 并跳转到读取 SD 卡函数的入口点; 读取 SD 卡函数执行完后会跳回 PC+8 的位置, 此时 Kernel 已被加载到指定位置, 再次使用 jal 跳转至 Kernel 的 main 函数入口点即可。

(3) Bootblock 如何调用 SD 卡读取函数

由资料可知读取 SD 卡函数 read_sd_card() 的入口点位于内存的 0x8007b1cc 处, 需要 3 个参数: 读取的目标地址、SD 卡内部的偏移量和要读取的字节数。读取的目标地址设为 0xa0800200, 为 Kernel 的入口地址; SD 卡内部的偏移量即 0x200, 是紧贴 Bootblock 的第二个扇区; 要读取的字节数在任务三中可以认为是一个扇区 (因为 Kernel 较小), 也可以通过.\ createimage 来确定读取的字节数。后续应当根据制作 image 时写入的 Kernel 大小 os_size 而定。根据 MIPS 调用约定，3 个参数应按顺序传入 4、5、6 号寄存器, 之后使用 jal 指令将 PC+8 存入 31 号寄存器并跳转到入口点 0x8007b1cc 即可。

(4) Bootblock 如何跳转至 kernel 入口

kernel_main 函数的地址为 0xa0800200, 使用 jal 指令将 PC+8 存入 31 号寄存器, 并跳转到该地址即可。

(5) 在设计、开发和调试 bootblock 时遇到的问题和解决方法

第一个问题是在做 bonus 题的过程中，对于使用 j 跳转指令还是 jal 跳转指令有些困惑，但后来在同学的提示下解决了这个问题。

另一个问题是后来考虑到内核的大小应该是可以变化的，所以稍微改进了一下，将 os_size 的值存放在了 bootblock 中的 0x1f0 处。

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

Bootblock 和 Kernel 都是经过汇编器、链接器编译的 ELF 可执行文件, 拥有完整的可执行文件头信息、块信息,Createimage 将 Bootblock 和 Kernel 中的可加载块分别读取出来, 并依次写入 image 文件, 因此 image 文件不包含任何头信息、块信息, 而是纯指令或数据组成的二进制文件。

(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 编译后都是 ELF 可执行文件, 可以读取 ELF 文件头, 从 e_phoff 域可知道段头的偏移,e_phnum 域可知道段头的数量; 之后跳转到段头偏移处, 遍历所有段头, 从 p_type 域是否为 PT_LOAD 可知该段是否能被加载,p_offset、p_vaddr 域则分别标注了该段的物理地址和虚拟地址,p_filesz、p_memsz 域分别标注了该段的物理大小和虚拟大小。

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

在 Createimage 执行时即统计 Kernel 二进制文件的大小, 写入 Bootblock 二进制文件中某预定位置 (我选取了 0x1f0, 因为 bootblock 在 0x80 之后全为 0, 可以用来存放数据) , Bootblock 运行时从该预定位置读取一个字, 即可知道 Kernel 二进制文件的大小。

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

第一个问题是开始时不知道如何对代码进行调试，后来同学提醒可以用 hexdump 来查看二进制码，mipsel-linux-objdump 来查看反汇编，调试起来就轻松多了。

第二个问题是之前一直将内核当作固定大小进行写入，后来在助教老师的提示下明白了 os_size 参数的含义，想清楚了如何将 os_size 写入 bootblock 的一个固定位置，以便 read_sd_card 进行调用，实现可变大小的内核的搬运。

3　关键函数功能

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）

第一个很关键的问题是在做 bonus 题的时候遇到的，这里非常巧妙地添加了一个死循环以保证 kernel 只运行了一遍,避免 kernel 中的函数不断返回初始地址而重复运行。我通过 mipsel-linux-objdump 来查看反汇编后发现，这是因为 kernel 最后会返回 31 号寄存器中的值，而由于在 bootblock 中给 31 号寄存器赋值为 0xa0800000，所以 kernel 会不断跳到 kernel_main 的位置运行。

```
//kernel.c
void __attribute__((section(".entry_function"))) _start(void)
{
    //Call PMON BIOS printstr to print message "Hello OS!"
    char *s = "Hello OS!\r\n";
    asm(
        "addi $a0, %0, 0x0 \t\n"
        "jal 0x8007b980 \t\n"
        :
        :"r"(s)
    );
    for(;;)
    {
        ;
    }
    return;
}
```

另外考虑到 #define PORT 0xbfe48000 的宏定义，我还尝试了另外两种写法，也顺利通过测试。

```
void __attribute__((section(".entry_function"))) _start(void)
{
    // Call PMON BIOS printstr to print message "Hello OS!"
    char *s = "Hello OS!";
    unsigned long port = PORT;
    while (*s) {
        *(unsigned char*)port = *s;
        s++;
    }
    return;
}
```

```
void printstr(char *s)
{
    unsigned long port = PORT;
    while (*s) {
        *(unsigned char*)port = *s;
        s++;
    }
}
```

```
9  void __attribute__((section(".entry_function"))) _start(void)
10 {
11     // Call PMON BIOS printstr to print message "Hello OS!"
12     printstr("Hello OS!\r\n");
13     return;
14 }
```