Project1 Bootloader 设计文档（RISC-V 版）

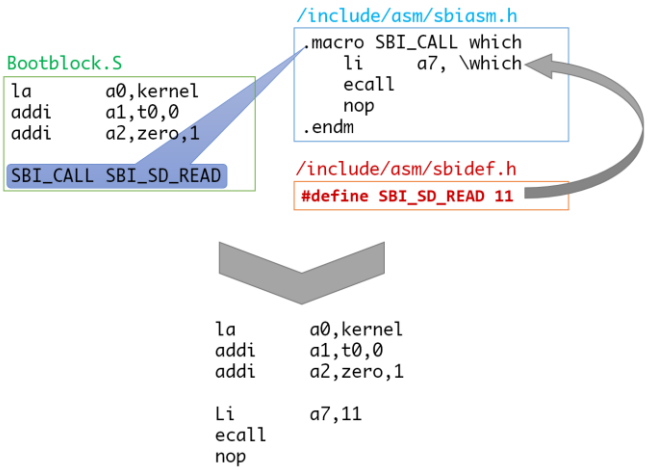
中国科学院大学
王嵩岳
2020 年 9 月 22 日

1. Bootblock 设计

在机器启动时，BIOS（PYNQ 板使用 BBL）将 SD 卡第一个扇区拷贝至内存相应位置，并将操作权交给 Bootblock。由于 Bootblock 仅有一个扇区（512B）的大小，所以我们不可能希望 BBL 将整个操作系统内核拷贝至内存。于是 **Bootblock 主要完成的工作即将剩余的操作系统内核从 SD 卡其它扇区中全部拷贝过来**。

SD 卡读取函数是 BIOS（BBL）提供的基本函数功能之一。在 Bootloader 中调用 SD 卡读取函数，实际上是做了一次相应的系统调用（RISC-V spec 中称为环境调用）。在本实验框架中，已经通过 sbi 驱动封装好了 SD 卡读取函数。因此我们只需要先根据 RISC-V ABI 和环境调用约定进行传参，而后使用 SBI_CALL 叫环境调用。

为了清晰地表达这个过程，我制作了图表 1。传参后，本质上首先是对 SBI_CALL 做了一个宏替换（汇编 marco 伪指令），将 SBI_CALL FUNC_NAME 转化为相应的环境调用。由 RISC-V 环境调用约定，环境调用首先须向 a7 寄存器传递环境调用号，然后执行 ecall 进行环境调用。环境调用号以宏定义的形式定义在 sbidef.h 中，将宏替换参数 which 进行替换即可。



图表 1 bootloader 中 BIOS 函数的调用

内核拷贝完成后，bootloader 的任务完成，需把控制权交给 kernel。这一步执行也很简单，只需让 PC 跳转至 kernel 的入口地址即可（S-Core 为 0x50201000，C-Core 是 0x50201000，其实是跳到 head.S 的入口_start）。这里我采用如下方法：先将 kernel_main 的地址加载到寄存器中，再用 jr 进行跳转。这看似多此一举，实则是为了防止使用 j、call 等指令而导致被编译器编译为相对跳转（对 S-Core 来说不会有问题，但是对 C-Core 来说，由于搬运导致相对位置变化，最好采用绝对跳转保证正确性）。

```
60      // jump to kernel_main
61      la t1,kernel_main
62      jr t1
```

图表 2 bootloader 跳转至 kernel

关于拷贝 kernel 的过程中，为支持多扇区 kernel 的拷贝，在 createimage.c 中我们已经把 kernel 所占扇区数存放在 0x502001fc 处，占用两字节。这里使用 lh 指令读出。

```
25      // save the size of kernel from memory in t0
26      la t0,os_size_loc
27      lh t0,0(t0)
```

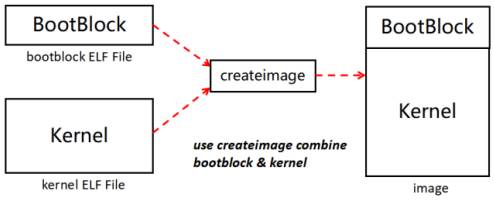
图表 3 读取 kernel 所占扇区信息

为遵守 RISC-V ABI 约定，避免滥用寄存器甚至影响系统调用传参，这里我们均使用 t0~t7 临时寄存器。

2. Createimage 设计

目前大多数机器二进制文件采用 ELF 格式存储，它包含了程序各段的信息。我们在操作系统中运行程序时，操作系统会通过解析 ELF header，进而读取各段表头、程序段，将各个段映射摆放入相应的内存区块。

而我们在 Bare Metal 上跑我们自己写的操作系统时，没有程序或部件帮助我们将 kernel 二进制文件解析而后正确地摆放至内存。因此我们需要在编译完内核后直接对此二进制文件进行解析，写入镜像，将各段摆放在相应的位置，这样当 kernel 被 bl 拷贝至内存后，就可以直接运行了。因此 createimage 的功能主要就是把 bootblock 和 kernel 两个 ELF 文件各段展开，摆放至镜像 image 的相应位置。另外，还需要把 kernel 实际占用的扇区数记录下来，放置在 0x502001fc 处。



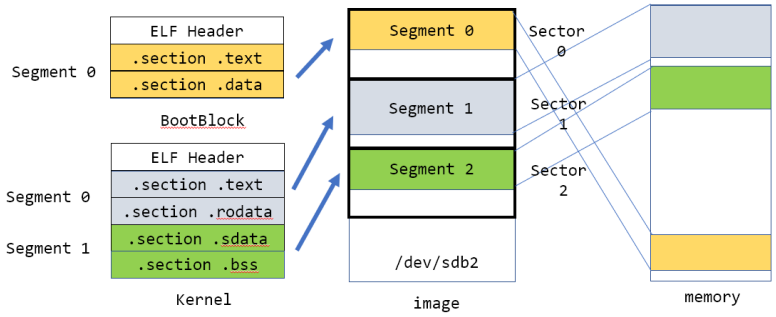
图表 4 createimage 的功能（S-Core）

获取 bootblock 和 kernel 两个二进制文件的中各段的位置和大小是通过两级解析完成的。首先对每个文件来说，我们首先解析其 ELF header，得知其有多少个程序头，以及每个程序头的位置（更准确的说是偏移）。然后再对每个程序头读出其数据，写入 fp 中，再通过 write_segment 函数写入镜像 img。

在实际开发时，我结合 GNU/RISC-V 工具链（主要是 objdump），分析其各段的结构。然后在 createimage.c 中添加解析打印信息（图表 5），得到了以下的结构对应图（图表 6）。经过统计，我的 kernel 占用了两个扇区，两个 segment 分别是（代码段、常量区）和（数据段、全局未初始化段）。

```
77          read_ehdr(&ehdr, fp);
78          printf("0x%04lx: %s\n", ehdr.e_entry, *files);
79          /* for each program header */
80          printf("File: %s has %d sections.\n", *files, ehdr.e_phnum);
```

图表 5 添加解析打印信息



图表 6 我的实现过程描述（C-Core）

为了让 bootblock 能够从 0x502001fc 获取到 2 字节的扇区数目信息，我们需要对写入的扇区数目进行统计，并完善 **write_os_size** 函数把统计值写入相应地址中。在这里我对框架中的 first 变量进行了功能修改，让其记录下一个应写入的 img 扇区。另外，nbytes 记录共写入的字节数（注意：由于各个 segment 中剩余区块也要填充为 0，因此我把这部分也计入 nbytes 中，这意味着 nbytes 一定是 512 的整数倍。**另外，由于 bootblock 占用一个扇区，故 kernel_size 应当减 1**）。

在写入 img 对应位置的时候还需注意，我们使用的内存是按 little-endian 编码的，因此写入时需要将高字节和低字节调换一下再写入。

```
125 static void write_os_size(int nbytes, FILE * img)
126 {
127     // calculate the kernel size
128     int kernel_size = (nbytes >> 9) - 1;
129     // calculate the pointer
130     fseek(img, (long)(OS_SIZE_LOC), SEEK_SET);
131     // save kernel size to memory (little endian), two bytes number
132     char data[2] = {kernel_size & 0xff, (kernel_size & 0xff00) >> 8};
133     fwrite(data, 1, 2, img);
134     printf("Kernel Size: %d\n", kernel_size);
135 }
```

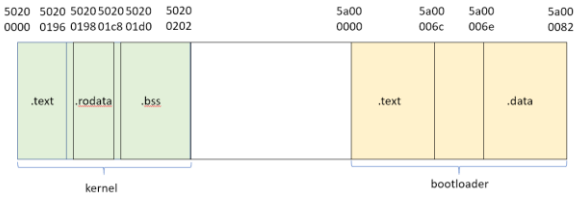
图表 7 写入 kernel 占用扇区数

3. A-Core/C-Core 设计

1. BootBlock 重定位

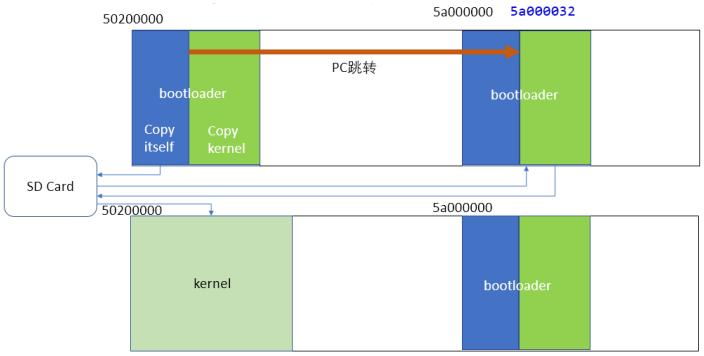
在某些实际场景下，如嵌入式系统中，实际可用内存空间并不充裕。因此我们希望通过将 Bootblock 重定位至其它位置，将 kernel 加载到 PC 复位的起始地址，以此节约 512B 内存空间。

根据内存结构布局，我选择将 Bootblock 重定位至 0x5a000000 处。这是实际实现后，我利用 GNU/RISC-V 工具链中 objdump 查看的结果。



图表 8 搬运后的内存布局

实现重定位需要对 bootblock 的功能进行重新设计，我将 bootblock 的功能主要分为两大块，第一部分是再拷贝 bootblock，第二部分是拷贝 kernel。



图表 9 C-core BL 设计简介

（1） **提前读取 0x502001fc 处的 kernel 所占扇区数：**在实现中，为避免内存覆盖的问题，我先将 size 存入寄存器 t0 中。但是其实这一步不是必须的，完全可以在后面读取。

```
25      // save the size of kernel from memory in t0
26      la t0,os_size_loc
27      lh t0,0(t0)
```

图表 10 读取扇区数

（2） **将 bootblock 拷贝至新地址 0x5a000000：**这个动作有多种实现方式，可以直接进行 memcpy，也可以再将 SD 卡的第一个扇区直写在新地址处。这里我采用后者，实现较为简单。

```
29      // copy new bootblock to new position 0x5a000000
30      la a0,bl_new
31      addi a1,zero,1
32      addi a2,zero,0
33      SBI_CALL SBI_SD_READ
```

图表 11 BootBlock 重定位

（3） **跳转至新 Bootblock 的拷贝后的指令执行：**如前所述，我们将 bootblock 的功能分成两块，现在需要跳转到 bootblock 的“拷贝 kernel 块”执行。这里我通过 GNU/RISCV 工具链中的 objdump 工具查看这条指令相对入口地址具体的偏移量。

```
stu@ucas-os:~/UCAS_OS/Project1-Bootloader$ riscv64-unknown-linux-gnu-objdump -d bootblock

bootblock:     file format elf64-littleriscv


Disassembly of section .text:

0000000050200000 <_ftext>:
    50200000:   0ff0000f                fence
    50200004:   502002b7                lui     t0,0x50200
    50200008:   1fc2829b                addiw   t0,t0,508
    5020000c:   00029283                lh      t0,0(t0) # 50200000 <_ftext>
    50200010:   5a000537                lui     a0,0x5a000
    50200014:   00100593                li      a1,1
    50200018:   00000613                li      a2,0
    5020001c:   48ad                    li      a7,11
    5020001e:   00000073                ecall
    50200022:   0001                    nop
    50200024:   0000100f                fence.i
    50200028:   5a000337                lui     t1,0x5a000
    5020002c:   0323031b                addiw   t1,t1,50
    50200030:   8302                    jr      t1
    50200032:   4529                    li      a0,10
    50200034:   4885                    li      a7,1
    50200036:   00000073                ecall
```

图表 12 使用 objdump 查看中间指令偏移量

可以看出，黄线处是拷贝完成后的第一条指令，因此我们应让 PC 跳转至 0x5a000000+0x32 处执行。

```
9   .equ bl_start, 0x5a000032
38      // jump to next instr in new bootblock
39      la t1,bl_start
40      jr t1
```

图表 13 跳转到重定位后的下一条指令

（4） **拷贝 kernel 至 0x50200000**：这里需要注意的是，kernel 覆盖了最早的 bootblock 的位置，为确保 cache 内存一致，这里需要用 fence.i 指令刷新一下 I-Cache。

```
13  .equ kernel, 0x50200000

51      // task2 call BIOS read kernel in SD card and jump to kernel start 0x50200000
52      la a0,kernel
53      addi a1,t0,0
54      addi a2,zero,1
55      SBI_CALL SBI_SD_READ
56
57      // avoid i cache inconsistent with memory
58      fence.i
```

（5） **跳转至_start@kernel**：需要注意的是，由于我们编译的时候给 bootblock 的起始地址是 0x50200000，给 kernel 的入口地址也是 0x50200000，故若直接使用伪指令 j 指令进行跳转，很可能编译器选择为相对跳转。然而实际上这条指令是在 0x5a000000 以后被执行的，使用相对跳转显然会得到错误的结果。因此这里我选择先将地址加载进寄存器，再利用 jr 指令绝对跳转。

```
60      // jump to kernel_main
61      la t1,kernel_main
62      jr t1
```

图表 14 跳转至 main@kernel

2. Createimage 支持大 kernel 的合并

实际上，如果按照所给的代码框架进行编写，不仅可以支持任意扇区的 kernel，也可以支持多于两个二进制文件的镜像制作。功能逻辑展现在下图中：

```
STEP 1: 以写模式打开img镜像
STEP 2: 遍历所有待合并的文件（对argv中的每一个参数）
            以读模式打开文件
            读取ELF Header，获取程序段数量及初始位置
            对每一个程序头
                读取段内容
                写入镜像扇区
                统计写入的size
            关闭文件
STEP 3: 将kernel的size写入img镜像偏移0x1fc处
STEP 4: 关闭镜像img文件
```

图表 15 createimage.c 的逻辑功能

其中，读取 ELF 文件信息时，对 ELF Header 的解析较为容易，只需全部读出写入 ehdr 即可。而对程序头的解析，其地址需要在程序段偏置（ehdr.e_phoff）的基础上再加一项偏置（ph * ehdr._ephentsize），以确保能读到程序区的每一个段。

```
94  static void read_ehdr(Elf64_Ehdr * ehdr, FILE * fp)
95  {
96      if(!fread(ehdr, sizeof(Elf64_Ehdr), 1, fp))
97          printf("[ERROR] Failed to read ehdr.\n");
98  }
99
100 static void read_phdr(Elf64_Phdr * phdr, FILE * fp, int ph,
101                       Elf64_Ehdr ehdr)
102 {
103     fseek(fp, (long)(ehdr.e_phoff + ehdr.e_phentsize * ph), SEEK_SET);
104     if(!fread(phdr, sizeof(Elf64_Phdr), 1, fp))
105         printf("[ERROR] Failed to read phdr header");
106 }
107
```

图表 16 读取 ELF 文件信息

在写入镜像文件时，根据讲义，实际在内存（镜像）中需要使用 memsize 进行大小的计算，filesize 和 memsize 之间的部分填 0。另外需要注意的是，当一个 segment 需占用非整数个扇区时，我们需要向上取整，并将空余地方清空为 0。这里使用单字节数组（char 数组）进行操作。

如果在 head.S 中已经将 bss 段清空，那么使用 filesize 计算也是可行的。注意：这里的 first 的含义有所变化，我用之记录已经写了多少扇区。

```
static void write_segment(Elf64_Ehdr ehdr, Elf64_Phdr phdr, FILE * fp,
                          FILE * img, int *nbytes, int *first)
{
    size_t file_size = phdr.p_filesz;
    size_t offset = phdr.p_offset;
    size_t mem_size = phdr.p_memsz;
    // calculate the size (x * 512B)
    int size = (mem_size + 511) / 512 * 512;
    char data[size];
    fseek(fp, (long)(offset), SEEK_SET);
    fread(data, file_size, 1, fp);
    fseek(img, (long)(*first * 512), SEEK_SET);
    fwrite(data, size, 1, img);
    // refresh nbytes
    *nbytes += size;
    *first += size / 512;
}
```

图表 17 将段写入镜像

4. 关键函数功能

关键代码已经在上面的分析中展示，在此不做赘述。

参考文献

[1] P1_RISCV_Guidebook
[2] RISC-V-Reader-Chinese-v2. CRVA, ICT
[3] ELF 文件格式解析 (https://zhuanlan.zhihu.com/p/52571826)
[4] 汇编语言 2020 春季. 第十五讲. ELF 文件格式

■