Project 1 Bootloader 设计文档

中国科学院大学
郑旭舟
2020 年 9 月 28 日

1. Boot block 设计

1.1. Boot block 主要完成的功能

Boot block 主要需要让 BIOS 跳转到 Boot Loader，使 Boot Loader 读取操作系统的内核（kernel）并将其加载到内存，最终跳转到 Kernel 运行。

1.2. Boot block 如何调用 SD 卡读取函数

将 SD 卡读取函数 void sd_card_read(void *dest, uint32_t offset, uint32_t size) 在开发板上的地址存入寄存器 t0，这个函数需要的三个参数分别是内容的目的地址、源在 SD 卡上的偏移位置和需要移动的源内容大小，将三个参数分别用 load 指令装入寄存器 a0 ~ a2，然后使用 jal 指令跳转到 t0 寄存器指示的地址，即可调用该函数。

1.3. Boot block 如何跳转至 kernel 入口

在完成 kernel 装载之后，将 kernel 在板上的地址存入寄存器 t0，然后使用 jal 指令跳转到 t0 寄存器指示的地址，就完成了到 kernel 入口的跳转。

1.4. 任何在设计、开发和调试 Boot block 时遇到的问题和解决方法

1.4.1. load 指令的选择

在调用 SD 卡读取函数的时候需要在寄存器中事先传入三个参数，但是一开始想当然地全都用 ld 指令，然后失败了。

因为这三个参数的类型不同，所以应该选择对应种类的 load 指令：用 la（load address）加载 kernel 的地址，用 li（load immediate）加载 SD 卡上 kernel 位置的字节偏移量，用 li 加载固定大小的小核。在后期需要适配大核的时候，需要间址寻址并使用 lw（load word）加载核的大小。

2. Create image 设计

2.1. Boot block 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 文件之间的关系

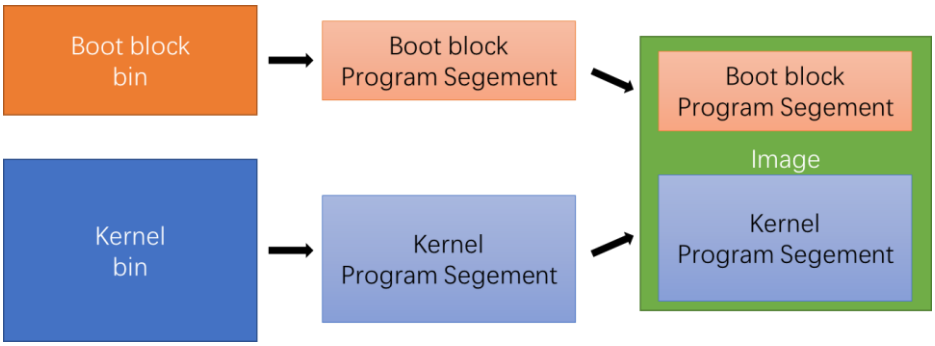

图 1 Boot block, Kernel 二进制文件与 image 之间的关系

如图所示，Create image 工具将 Boot block 的 ELF 文件中的程序段和 Kernel 的 ELF 文件中的程序段组合起来，成为写入 SD 卡的 Image 文件。

2.2. 如何获得 Boot block 和 Kernel 二进制文件中可执行代码的位置和大小，你实际开发中从 kernel 的可执行代码中拷贝了几个 segment?

从 ELF 文件中的 Ehdr（ELF Header）部分可以找到 Phdr（Program Header）的偏移位置、个数和单位大小，在 Phdr 可以找到对应程序段二进制可执行代码的位置和大小。

二进制可执行代码的代码段大小有 file_sz 和 mem_sz 两项，这两个分别是可执行代码的数据大小和代码运行时占据内存的大小，后者通常大于前者，根据实际情况，写 Create image 工具的时候采用 mem_sz，避免运行时溢出出错。

在实际开发中，从 kernel 的可执行代码中拷贝的 segment 个数由 Ehdr 中的 e_phnum 项决定。

2.3. 如何让 Boot block 获取到 Kernel 的大小，以便进行读取

Create image 时，将 kernel 的大小写在 image 的一个固定位置。

相应地改写 Boot block，使其在加载 kernel 前预先在寄存器 a3 中存入通过间址读出的 kernel 大小，而不是像实验 1 要求的那样存入一个预设的 kernel 大小。

2.4. 任何在设计、开发和调试 Create image 时遇到的问题和解决方法

2.4.1. 对读取多个程序段的支持

一开始为了简便，认为使用的是小核，并只读取一个程序段，但是很多 kernel 都不止一个程序段，只读取一个程序段不能将 kernel 完整地拷贝到 image 中。

将原有的读取单个程序段的代码放进 Create image 函数读写单文件的一个 for 循环中，循环变量设置为 e_phnum，并对代码中的读取偏移量等参数作相应修改，即可支持读取多个程序段。

2.4.2. kernel 的拷贝出错

写完 Create image 后的调试中，出现了无法正常执行 kernel 的情况，报错显示 TLB miss，分析后认为是写 kernel 大小时对文件流指针重定向的处理出了问题。

原来的处理方式是在每一次添加文件时都在 image 的固定位置 OS_SIZE_LOC 写一遍 kernel size，这样保留到最后的 kernel size 就是正确的 size. 选取的 OS_SIZE_LOC 是 boot block 所在扇区的 0x1f0 位置，并非扇区的末尾。

在实际处理中，由于没有重定向文件流指针，下一次的 write 中没有从 boot block 后的扇区开始写，而是从 OS_SIZE_LOC 之后开始写，但 boot block 的 kernel 位置是写死的。所以，boot block 执行 jal kernel 指令时，跳转到的位置不是 kernel 的入口地址，而是 kernel 中间的一条指令，因此 os 的行为会出错。

分析后决定优化写 size 的函数。维护全局变量 KN_flpsz 记录 kernel 大小，在所有程序段都写入镜像之后，将 KN_flpsz 写入 image 中，这样程序段就写在了正确的地方，并且也优化了执行逻辑和效率。

3. A-Core/C-Core 设计（可选）

本设计未实现重定向功能，但实现了大核加载。

实现大核加载的核心在于 kernel 大小必须是可变的，这就要求将 kernel 大小在 create image 的时候就写进 image 中，因此，create image 函数中需要添加 write_os_size 函数来完成该功能，size 的大小是所有写入 image 的程序段大小的总和。

因为设置 size 存在四字节变量内，为了防止核过大溢出，记录的 size 是扇区数而不是字节数，对应地在 boot block 中将扇区数左移 9 位即可得到字节数。

4. 关键函数功能

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）

4.1. 处理命令行参数

```
for (int j = 1; j < argc; j++)
{
    if ((*argv)[0] == '-' && (*argv)[1] == '-') // right fmt
    {
        // option
        if ((*argv)[2] == 'e')
            options.extended = 1;
        else if ((*argv)[2] == 'v')
            options.vm = 1;
        else
        {
            printf("Error: no such Arg!\n");
            exit(1);
        }
        argv++;
        continue;
```

```
    }
    else if ((*argv)[0] == '-' || (*argv)[1] == '-') // wrong fmt
    {
        printf("Error: wrong format for args! Example: %s\n", ARGS);
        exit(1);
    }
    // files
    cnt++;
}
```

这段函数处理了命令行参数选项（默认用户只能将参数放在文件前面的位置），并且由于循环变量 j 的大小设置为 0 到 argc，因此 argv 在处理文件名时不后移也不影响程序的正确性。

4.2. 将一个文件的全部程序段写入 image

```
// loop: write every psg
    for (int ph = 0; ph < b_ehdr->e_phnum; ph++)
    {
        // 1 -> more psg
        Elf64_Phdr *b_phdr = (Elf64_Phdr *)malloc(sizeof(Elf64_Phdr));
        read_phdr(b_phdr, fp, ph, b_ehdr);

        // [-extended] prtf every psg info
        if (options.extended)
            printf("0x%x: %s\n\t\toffset:0x%x\tvaddr:0x%x\n\t\tfilesz:0
x%x\tmemsz:0x%x\n", (unsigned int)b_phdr->p_vaddr, files[i], (unsigned
int)b_phdr->p_offset, (unsigned int)b_phdr->p_vaddr, (unsigned int)b_ph
dr->p_filesz, (unsigned int)b_phdr->p_memsz);

        // write psg in image
        write_segment(b_phdr, fp, image);
        free(b_phdr);
    }
```

这段函数读取当前文件的全部程序头，打印所有的程序段信息，并且将全部程序段中的可执行代码写入 image 中。

4.3. 处理命令行参数指定的全部文件

```
for (int i = 0; i < nfiles; i++)
{
    // fopen
    FILE *fp = fopen(files[i], "r+");
    if (!fp)
    {
        printf("Error: no such file to open!\n");
        exit(1);
```

```
    }

    // read hdrs
    Elf64_Ehdr *b_ehdr = (Elf64_Ehdr *)malloc(sizeof(Elf64_Ehdr));
    read_ehdr(b_ehdr, fp);
    // loop: write every psg
    for (int ph = 0; ph < b_ehdr->e_phnum; ph++) ...
    // free&close
    free(b_ehdr);
    fclose(fp);
}
```

这段函数依次处理命令行参数指定的文件：检查文件是否存在，通过解析文件头和程序头将对应可执行代码段写入 image 中。

参考文献

[1] Linux 的 elf.h 代码：https://github.com/torvalds/linux/blob/master/include/uapi/linux/elf.h

■