Project 1 Bootloader设计文档

中国科学院大学

郑旭舟

2020年9月28日

# Boot block设计

## 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。

## 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。

## 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。

## 任何在设计、开发和调试Boot block时遇到的问题和解决方法

### 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。

# Create image设计

## 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

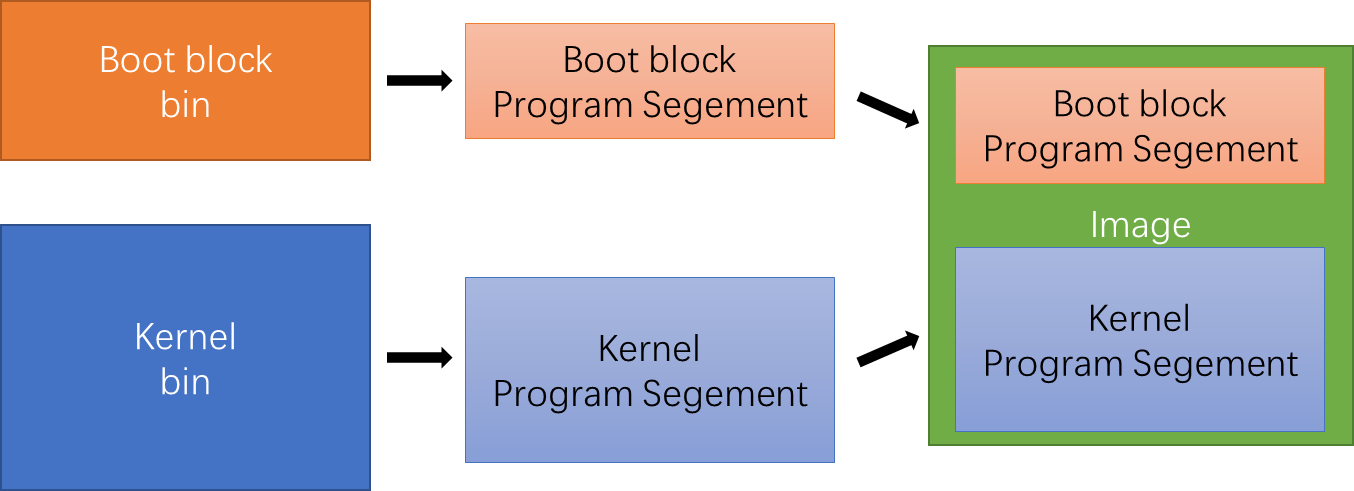


图 1 Boot block, Kernel二进制文件与image之间的关系

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。

## 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。

## 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。

## 任何在设计、开发和调试Create image时遇到的问题和解决方法

### 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。

### 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。

# 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。

# 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）

## 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。

## 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:0x%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\_phdr->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。

## 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

▄