Project1 Bootloader 设计文档

中国科学院大学
[姓名] 蔡洛姗
[日期] 2020.9.27

1. Bootblock 设计

（1）Bootblock 的功能：Bootblock 最主要的任务是将操作系统的 kernel 部分代码从存储设备（SD 卡），搬运到内存中，并且移交控制权，跳转到 kernel 入口开始执行。

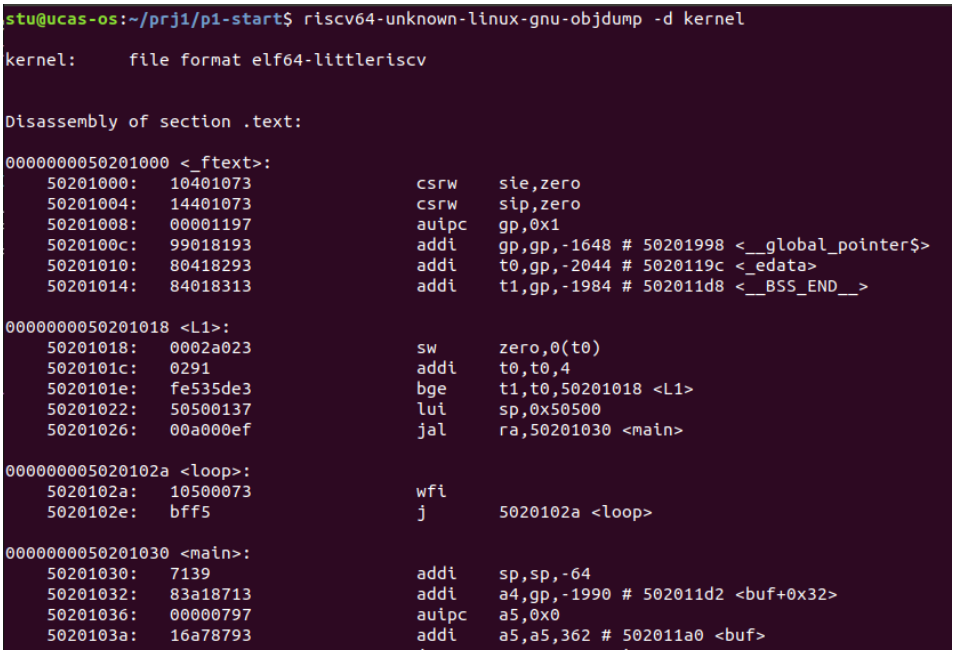
（2）Bootblock 如何调用 SD 卡读取函数：在 RISC-V 代码中，老师已经封装好了 SBI_SD_READ 函数，可以通过 SBI_CALL 调用，而我们需要做的，就是把函数所需的三个参数：mem_address（目标内存地址），num_of_blocks（读取扇区个数），block_id（扇区号）放入相应的寄存器 a0,a1,a2 中即可。例如：

```
la        a0, kernel          #mem_addr
li        a1, 1               #num_of_blocks
li        a2, 1               #block_id
SBI_CALL SBI_SD_READ
```

（3）Bootblock 如何跳转至 kernel 入口：通过分析 riscv.lds 文件就可以知道，在链接过程中内核的入口函数被放在了内核文件的开头，也就是内存地址为 0x50201000 处，所以只需要 Bootblock 在执行完其他所有任务后，用一条 call kernel_main（或 la t0 kernel_main, jr t0）指令跳转到 kernel 入口的地址处开始执行即可。

（4）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法：

在开始执行 kernel.c 文件之前，还需要利用 head.S 文件的内容为 C 语言的执行创建环境：清空 bss 段内存，设置栈指针。但是之前又说 Bootblock 最后要跳转到 kernel.c 的入口，那么 head.S 的内容到底怎样才会被执行呢？在编译 kernel.c 文件之后，我用 objdump 查看了 kernel 可执行文件中的内容，发现其实在 kernel 的开头已经包含了 head.S 的内容，如下图。



```
stu@ucas-os:~/prj1/p1-start$ riscv64-unknown-linux-gnu-objdump -d kernel

kernel:     file format elf64-littleriscv


Disassembly of section .text:

0000000050201000 <_ftext>:
    50201000:   10401073                csrw    sie,zero
    50201004:   14401073                csrw    sip,zero
    50201008:   00001197                auipc   gp,0x1
    5020100c:   99018193                addi    gp,gp,-1648 # 50201998 <__global_pointer$>
    50201010:   80418293                addi    t0,gp,-2044 # 5020119c <_edata>
    50201014:   84018313                addi    t1,gp,-1984 # 502011d8 <__BSS_END__>

0000000050201018 <L1>:
    50201018:   0002a023                sw      zero,0(t0)
    5020101c:   0291                    addi    t0,t0,4
    5020101e:   fe535de3                bge     t1,t0,50201018 <L1>
    50201022:   50500137                lui     sp,0x50500
    50201026:   00a000ef                jal     ra,50201030 <main>

000000005020102a <loop>:
    5020102a:   10500073                wfi
    5020102e:   bff5                    j       5020102a <loop>

0000000050201030 <main>:
    50201030:   7139                    addi    sp,sp,-64
    50201032:   83a18713                addi    a4,gp,-1990 # 502011d2 <buf+0x32>
    50201036:   00000797                auipc   a5,0x0
    5020103a:   16a78793                addi    a5,a5,362 # 502011a0 <buf>
```

因此事实上，kernel 的入口就是 head.S，当 Bootblock 跳转到 0x50201000 时，首先执行的就是建立运行环境的程序，然后才进入 main 函数开始运行操作系统内核的真正内容。

2. Createimage 设计

（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 文件是由 Bootblock 编译后的二进制文件和 Kernel 编译后的二进制文件去掉文件头、程序头、段表头等内容，将其每一个段的内容扩展成在内存中所占大小之后，每个文件再按扇区补齐拼接而成的。

（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，你实际开发中从 kernel 的可执行代码中拷贝了几个 segment？

首先需要读取二进制文件的文件头内容，然后根据文件头中的 e_phoff（程序头的偏移地址）定位到程序头的位置，最后按照程序头中指示的 segment 地址和大小，以及文件头包含的 segment 个数，就可以计算可执行代码的位置和大小。

实际运行中，打印出来的信息如下图：

```
stu@ucas-os:~/prj1/p1-start$ make all
gcc createimage.c -o createimage -ggdb -Wall
chmod +x ./createimage
./createimage --extended bootblock kernel
0x50200000: bootblock
        segment 0
                offset 0x1000   vaddr 0x50200000
                filesz 0x5b     memsz 0x5b
                writing 0x5b bytes
0x50201000: kernel
        segment 0
                offset 0x1000   vaddr 0x50201000
                filesz 0x19c    memsz 0x1d8
                writing 0x1d8 bytes
        segment 1
                offset 0x0      vaddr 0x0
                filesz 0x0      memsz 0x0
                writing 0x0 bytes
os_size: 1 sectors
```

可以得知从 kernel 的可执行代码中拷贝了 2 个 segment。

（3）如何让 Bootblock 获取到 Kernel 的大小，以便进行读取：

在生成 image 镜像文件的过程中，可以读取 kernel 二进制文件的程序头中每一个 segment 的大小并累加，结果就是 kernel 的大小，将其存放在一个特定的位置（image 文件的 0x1fc 处，在内存中对应地址 0x502001fc），Bootblock 在需要该信息时就可以从该地址直接加载数据。

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

不知道段内容如何读取后写入？？？

3. A-Core/C-Core 设计（可选）

（1）你设计的 bootloader 是如何实现重定位的？如果 bootloader 在加载 kernel 后还有其他工作要完成，你设计的机制是否还能正常工作？

在考虑这个问题时我首先想到的思路是：因为覆盖发生在读 SD 卡的过程中，被覆盖的内容只剩下最后一条跳转指令，能不能通过修改某个寄存器，使得执行完 SD 卡读取函数之后，程序直接跳转到 kernel 执行呢？我先是考虑了 pc 寄存器，但是询问过老师之后才了解

到 pc 寄存器是不可写的；随后我又想是不是调用子程序之后会把返回地址放在 ra 寄存器，但是老师告诉我 sbi_call 展开是一个 ecall 系统调用，返回地址存在 mepc 中，这个寄存器只有 m 态可以读写，目前系统还是在 s 态运行的，无法读写 mepc 寄存器。而且这种方法也并不通用，如果 bootloader 在加载 kernel 后还有其他工作要完成，那也是无法实现的。

更好的方法是在 Bootblock 开头先把 bootloader 拷贝到其它空闲的内存空间中，防止它被 kernel 覆盖，然后再把内核加载到 0x50200000，执行完 bootbloader 之后再跳转到 kernel 入口地址 0x50200000，这种方案就更加合理，而且能支持 Bootloader 其它工作正常完成。

（2）遇到的问题：

在拷贝完 bootloader 之后，应该怎样保证 bootblock 不再执行开头拷贝的一段代码呢？我一开始用的方法是：因为只有拷贝代码的过程会使 a0 寄存器赋上值 0x50500000，因此我猜测如果是第二次开始执行 bootblock 的开头时，a0 寄存器就会保存这个值，所以先判断 a0 寄存器是不是 0x50500000，如果是就跳转到剩余的代码部分执行，如果不是就说明是第一次，需要拷贝。但是在执行过程中却发现界面上循环打印 it's a bootloader！说明分支跳转没有成功。于是我使用 qemu 和 gdb 从 0x50500000 处开始，查看寄存器的值，发现 a0 寄存器已经被清零了，说明不能使用这个寄存器来暂存。或许使用其它寄存器是可行的，但我没有继续尝试这种方法，而是改用方法二：计算拷贝部分消耗了几条指令，在 bootloader 拷贝完成后直接跳转到剩余部分代码的起始地址开始执行。指令的计算使用 gdb 中的指令显示就很方便，真正 bootloader 起始地址为 0x5050001a。

4. 关键函数功能

（1）Bootloader 重定位：

```
//C-core:move the bootloader to 0x50500000
la              a0, bootloader      #mem_addr
li              a1, 1               #num_of_blocks
li              a2, 0                           #block_id
SBI_CALL SBI_SD_READ
la      t0, bootloader_start
jr              t0
```

（2）调用 SD 卡读取函数并跳转到 kernel：

```
// 2) task2 call BIOS read kernel in SD card and jump to kernel start
la      a0, kernel          #mem_addr
la      a3, os_size_loc
lh      a1, (a3)            #num_of_blocks
li      a2, 1               #block_id
SBI_CALL SBI_SD_READ
la      t0, kernel_main
jr              t0
```

（3）head.S 清空 bss 段、设置栈指针，跳转到 C 文件执行：

```
/* Clear BSS for flat non-ELF images */
la t0, __bss_start
la t1, __BSS_END__
L1:
sw   zero, (t0)
addi t0, t0, 4
```

```
        ble  t0, t1, L1

        /* setup C environment (set sp register)*/
        la sp, KERNEL_STACK
        /* Jump to the code in kernel.c*/
        call main
```

（4）合并 bootloader 和 kernel 二进制文件，生成镜像：

```
        static void create_image(int nfiles, char *files[])
        {
                int ph, nbytes = 0, first = 1;
                FILE *fp, *img;
                Elf64_Ehdr ehdr;
                Elf64_Phdr phdr;
                /* open the image file */
                if((img = fopen(IMAGE_FILE,"wb+"))==NULL){
                        printf("Can not open image file!\n");
                        return;
                }
                /* for each input file */
                 while (nfiles-- > 0) {
                        /* open input file */
                        if((fp = fopen(*files,"r"))==NULL){
                                printf("Can not open input file!\n");
                                return;
                        }
                 /* read ELF header */
                        read_ehdr(&ehdr, fp);
                        printf("0x%04lx: %s\n", ehdr.e_entry, *files);
                        /* for each program header */
                        for (ph = 0; ph < ehdr.e_phnum; ph++) {
                              if(options.extended == 1){
                                 printf("\tsegment %d\n",ph);
                             }
                        /* read program header */
                        read_phdr(&phdr, fp, ph, ehdr);
                        /* write segment to the image */
                       write_segment(ehdr, phdr, fp, img, &nbytes, &first);
                }
                fclose(fp);
                files++;
            }
        write_os_size(nbytes, img);
        fclose(img);
        }
```

将每个 segement 写入 img 文件：

```
static void write_segment(Elf64_Ehdr ehdr, Elf64_Phdr phdr, FILE * fp,
                          FILE * img, int *nbytes, int *first)
{
    //read a segment
    char *seg;
    seg = (char *)malloc(sizeof(char)*phdr.p_filesz);
    fseek(fp, phdr.p_offset, SEEK_SET);
    fread(seg, phdr.p_filesz, 1, fp);

    //write bootblock
    char pad[512]="";
    if(*first){
        fwrite(seg, phdr.p_filesz, 1, img);
        //padding to a sector
        if(512-phdr.p_filesz){
            fwrite(pad, 1, 512-phdr.p_filesz, img);
        }
        *first = 0;
    }else{
    //write kernel
        //fseek(img, *nbytes, SEEK_SET);
        fwrite(seg, phdr.p_filesz, 1, img);
        fwrite(pad, 1, phdr.p_memsz-phdr.p_filesz, img);
        *nbytes += phdr.p_memsz;
    }
}
```

参考文献

[1] “国科大操作系统研讨课任务书”，P1_RISCV_Guidebook
[2] 《RISC-V 手册》，翻译：勾凌睿、黄成、刘志刚 校阅：包云岗

■