Project1 Bootloader 设计文档

中国科学院大学
贾志杰
2020 年 9 月 23 日

1. Bootblock 设计

（1）Bootblock 主要完成的功能

使用 printstr 函数打印提示字符串 "It's bootblock!"。

将 kernel 代码从 SD 卡拷贝到内存中指定位置。

跳转到 kernel 的入口位置，启动 kernel。

（2）Bootblock 如何调用 SD 卡读取函数

以小核、无重定位的简单情况调用 read_sd_card 为例，依次把函数的三个参数：拷贝到内存中的地址、SD 卡的偏移位置、读取的数据大小，分别传入$a0、$a1、$a2 寄存器。然后把 read_sd_card 函数入口地址给临时寄存器$t1，最后使用 jalr 调用函数。

```
ld    $a0,kernel
daddi $a1,$0,0x200
daddi $a2,$0,0x200
ld     $t1,read_sd_card
jalr    $t1
.data
read_sd_card: .dword 0xffffffff8f0d5e10
kernel : .dword 0xffffffffa0800200
```

（3）Bootblock 如何跳转至 kernel 入口

使用寄存器间接跳转，先把 kernel 入口地址放入临时寄存器$t1，最后使用 jalr 跳转到 kernel 入口。

```
ld     $t1,kernel_main
jalr    $t1
.data
kernel_main: .dword 0xffffffffa0800000
```

（4）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

最开始的问题是对 mips64 汇编指令不太了解，通过查阅 mips64 指令手册解决这一问题。

另一个比较困难的问题是重定位中读取 PC 的值，指令集中提供了几种获取 PC 的指令比如 ADDIUPC 等，但是编译时报错识别不出该指令。之后设想通过 JAL 的技巧获取 PC，不过这一方法并不稳妥，对于乱序执行的 CPU 可能出现错误。

最后也没有一个很好的解决方法，一个妥协方案是避免读取 PC 值，比如通过 gdb 找出拷贝完 Bootblock 后要跳转过去的那条指令地址，把这个地址写死。当然这个办法缺少通用性，一旦 Bootblock 前面部分的代码有变动，就必须要重新确定这个地址。另一个方案就是采用 JAL 的技巧，至少在目前的实验平台上它的正确性还是能够保证。

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

Bootblock 编译出的 Bootblock.o、Kernel 编译出的 main.o，这两者都是 ELF 文件，遵循 ELF 文件格式。而 image 文件不是 ELF 文件，它是从 Bootblock.o 和 main.o 两个文件中分别抽取其中的程序段，再通过补 0 对齐处理组合而成的。

（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？

首先从 ELF 文件读取 ELF 文件头，从 ELF 文件头中找出程序头表的地址和程序头的数量和大小。然后依次读取每个程序头，在每个程序头中找到对应程序段的地址和大小。

实际开发中 Kernel 的可执行代码中只有一个 segment。

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

在创建 image 时，计算出 Kernel 实际占据的扇区数，然后把 Kernel 的大小写入一个空白地方，比如 Bootblock 段的最后（0x1fc），这样就可以在 Bootblock 里访问该位置读取 Kernel 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

没有问题。

3. A-Core/C-Core 设计

（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？

首先考虑重定位的问题在哪里，其实就是在 bootloader 调用 read_sd_card 函数加载 kernel 后，内存中 kernel 的指令覆盖了 bootloader 的指令，read_sd_card 函数执行完返回 bootloader 后执行 kernel 的指令导致出错。那么这就有两个思路来处理：一是让 read_sd_card 函数返回后不要回到最开始调用它的地方；二是在调用 read_sd_card 函数前先把 bootloader 拷贝到一个安全的地方。

方法一：

函数调用是通过$ra 寄存器存放返回地址的，我们使用了 jalr 调用 read_sd_card 函数，它隐含$ra=pc+8。但是现在已经没必要返回到 pc+8 的位置，因为后面的代码都被 kernel 覆盖了。所以我们可以直接操控$ra，使用 jr 伪装成 jalr，这样当 read_sd_card 函数执行完，他就会根据$ra 直接跳转到 kernel_main。

代码如下：

```
ld    $t1, read_sd_card
ld    $ra, kernel_main
jr    $t1
.data
kernel_main: .dword 0xffffffffa0800000
```

这个方法比较取巧，如果我们希望 bootloader 在加载 kernel 后还要完成其他工作，那么就不能使用这个方法，这时需要使用方法二。

方法二：

既然 read_sd_card 函数加载 kernel 后，内存中 kernel 的指令会覆盖 bootloader 的指令，那么我们可以在调用 read_sd_card 函数前先把 bootloader 的指令先拷贝到一个安全位置，然后从该位置继续执行 bootloader 的指令。因为在实现大核加载时我们已经知道 kernel 的大小，因此 0xffffffffa0800000 加上 kernel 的大小（按字节）就是安全位置。

可以将 Bootblock.S 的第 2 部分分为下面几个步骤：#read kernel size、#copy bootloader to the back of kernel、#jump to the copy of bootloader、#read kernel in SD card。其中关键问题是确定跳转到 bootloader 备份时需要跳到哪条指令，如果我们将整个 bootloader 段拷贝过去，那么只需要将跳转这条指令的地址加上 kernel 的大小再加 8 即可，于是问题转化为确定该条指令的 PC。具体的方法在前面 Bootblock 的设计中已经提及，这里不再赘述。

这个方法总的来说比较麻烦，但是由于把 bootloader 拷贝到了不会被覆盖的地方，在完成加载 kernel 后 bootloader 仍然可以继续正常执行。如果我们希望 bootloader 在加载 kernel 后还要完成其他工作，那么该方法可以保证这一需求。

4. 关键函数功能

请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）

Bootblock.S 中重定位部分

```
# 2) call BIOS read kernel in SD card
      #read kernel size
    ld   $t0, kernel_size_addr
    lhu  $t1, 0($t0)
    dsll  $t1, $t1, 9
      #copy bootloader to the back of kernel
    ld  $t0, kernel
    daddi $t2, $t0, 0
    daddi $a0, $t0, 0
    daddi $a0, $a0, 0x200
    dadd  $t0, $t0, $t1
    copy:
    ld  $t3, 0($t2)
    sd  $t3, 0($t0)
    daddi $t2, $t2, 8
    daddi $t0, $t0, 8
    bne $t2, $a0, copy
      #jump to the copy of bootloader
    jal read_pc
    read_pc:
    dadd  $ra, $ra ,$t1
    daddi $ra, $ra, 16
    jr $ra
```

Createimage.c 里的 write_segment 函数

需要注意其中 memsz 和 filesz 的区别，把两者之差部分填充 0。最后写完 segment 还要按扇区补 0 对齐。

```
static void write_segment(Elf64_Ehdr ehdr, Elf64_Phdr phdr, FILE *fp,
                          FILE *img, int *nbytes, int *first)
{
    char *segment = (char*)malloc(phdr.p_memsz);
    unsigned int write_offset;
    unsigned int padding_size;

    /* read segment from elf */
    memset(segment,0,phdr.p_memsz);
    fseek(fp,phdr.p_offset,0);
    fread(segment,1,phdr.p_filesz,fp);
    rewind(fp);

    /* calculate offset in image file */
    if(*first){
        write_offset = 0;
        *first = 0;
    }
    else{
        write_offset = SECTOR_SIZE + *nbytes;
        if(phdr.p_memsz % SECTOR_SIZE)
            *nbytes = *nbytes + ((phdr.p_memsz / SECTOR_SIZE)+1) * SECT
OR_SIZE;
        else
            *nbytes = *nbytes + phdr.p_memsz;
    }

    /* write segment to image */
    fseek(img,write_offset,0);
    fwrite(segment,1,phdr.p_memsz,img);
    free(segment);
    if(options.extended)
    printf("\t\twriting 0x%04lx bytes\n",phdr.p_memsz);

    /* pad 0 by sector alignment */
    if(phdr.p_memsz % SECTOR_SIZE){
        padding_size = SECTOR_SIZE - (phdr.p_memsz % SECTOR_SIZE);
        char *padding = (char*)malloc(padding_size);
        memset(padding,0,padding_size);
        fwrite(padding,1,padding_size,img);
```

```
        free(padding);
    }
    if(options.extended)
    printf("\t\tpadding up to 0x%04x\n",(*nbytes)+SECTOR_SIZE);
    rewind(img);
}
```

参考文献

[1] MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual

■