Project1 Bootloader 设计文档

中国科学院大学
葛忠鑫
2020/9/27

1. Bootblock 设计

（1）Bootblock 主要完成的功能

将操作系统代码搬运到内存。

（2）Bootblock 如何调用 SD 卡读取函数

$a0~$a7 寄存器是用于存放函数调用参数的，在调用 read_sd_card 函数时，将它的三个参数移动目的地址 dest，SD 卡读取 kernel 位置的偏移量 offset 和读取数据的大小 size 分别存入$a0，$a1 和$a2。未实现重定位时，要将内核加载到 Boot Loader 后面，所以 dest 为 0xffffffffa0800200（即 kernel 的值）；Boot Loader 有 512 个字节，存放在 SD 的第一个扇区，其后 kernel 相对 SD 卡的偏移量 offset 即为 0x200（=512）；在仅实现小核（一个扇区足够存放）加载的时候，只需读取一个扇区大小的 size=0x200 即可。

read_sd_card 所用参数准备好后，利用 jal 指令跳转至其所在地址执行。

```
1.      ld $a0, kernel
2.      li $a1, 0x200
3.      li $a2, 0x200
4.      ld $t0, read_sd_card
5.      jal $t0
```

（3）Bootblock 如何跳转至 kernel 入口

将 kernel 的起始地址 0xffffffffa0800200（kernel_main）写入临时寄存器，然后执行跳转指令。

```
1.      ld $t0, kernel_main
2.      jal $t0
```

（4）在设计、开发和调试 bootblock 时遇到的问题和解决方法

对 MIPS64 指令集不熟悉，不知道说明命令还可以用，什么命令需要扩展位宽。后面老师发的 MIPS64 指令手册解决了一些困惑，不知道的时候查指令集手册看看具有所需功能指令的具体用法。

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 文件由两部分组成，第一部分是 Bootblock 编译后的二进制文件中的程序段部分，需要存放在 SD 卡中的第一扇区，即 image 的前 512 比特；第二部分是 Kernel 编译后的二进制中的程序段部分，在 SD 卡中的第一扇区后面的一个或多个扇区，即 image 和 512 比特位置以后。

（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？

Bootblock 和 Kernel 二进制文件开头都是 ELF 头和程序头，ELF 头结构体中的 e_phoff 变量表示程序头里 ELF 头的偏移量，可以找到程序头的位置，程序头结构体中的 p_offset 和 p_filesz 分别表示可执行代码段的偏移量和大小。

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

将 kernel 的大小（双字节）写到第一个扇区的末尾的一个固定地址中，供 Boot Loader 执行时使用。

（4）在设计、开发和调试 createimage 时遇到的问题和解决方法

对文件读写操作不熟悉，需要反复查阅函数使用方法。

3. A-Core/C-Core 设计

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？

函数调用后的返回地址存放在$ra 寄存器中，可以通过修改 read_sd_card 函数的返回地址以实现重定位，即在跳转（jr）至 read_sd_card 前给$ra 寄存器写入正确的 kernel 的起始地址。具体代码如下：

```
1.      ld $a0, kernel
2.      li $a1, 0x200
3.      # li $a2, 0x200
4.      lw $a2, kernel_size
5.      dsll $a2, $a2, 9
6.      ld $t0, read_sd_card
7.      ld $ra, kernel_main
8.      jr $t0
```

其中 kernel 和 kernel_main 地址改为覆盖 Boot Loader 后的起始地址 0xffffffffa0800000，kernel_size 为存放在 Boot Loader 末端的内核大小地址 0xffffffffa08001fc。

如果加载 kernel 后还有其他工作，在上述方法下就不能正常工作了。如果有其他工作需要完成，可以先将 Boot Loader 内容拷贝至内存中 kernel 的后面（因为在这个可以加载大核的设计中 kernel 大小可以提前读出），然后根据机器指令找到拷贝后未执行的 Boot Loader 代码段的起始位置继续执行（也可以根据机器指令只拷贝未执行的代码部分），在执行完 Boot Loader 后将该块内存清空也可实现重定位以避免资源浪费。

重定位要注意修改 Makefile 中 main 的-Ttext 参数为 0xffffffffa0800000。

4. 关键函数功能

(1) write_os_size 函数

```
1. static void write_os_size(int nbytes, FILE *img){
2. char others[4] = {0x01, 0x00, BOOT_LOADER_SIG_1, BOOT_LOADER_SIG_2};
3.     int kernel_size = (nbytes - 1) / SECTOR_SIZE;
4.     others[0] = (char)kernel_size;
5.     others[1] = (char)(kernel_size >> 8);
6.     fseek(img, BOOT_LOADER_SIG_OFFSET - OS_SIZE_LOC, SEEK_SET);
7.     fwrite(others, 4, 1, img);
8. }
```

kernel_size 两个字节以小端序存放在 0xffffffffa08001fc 起始的两个比特内，方便 Boot Loader 使用 lw 指令直接取出。这里要注意根据 image 总的写入字节数 nbytes 计算 kernel 大

小时需要减去 Boot Loader 的那一个扇区。

(2) write_segment 函数

```
1.  static void write_segment(Elf64_Ehdr ehdr, Elf64_Phdr phdr,FILE *fp,
2.                            FILE *img, int *nbytes, int *first){
3.      // create a buffer to hold the image
4.      char * file = (char *)malloc(sizeof(char) * phdr.p_filesz);
5.      // clear the buffer
6.      memset(file, '\0', phdr.p_filesz);
7.      // read the file
8.      fseek(fp, phdr.p_offset, SEEK_SET);
9.      fread(file, phdr.p_filesz, 1, fp);
10.     // write the file to the image
11.     fseek(img, SECTOR_SIZE * (*first), SEEK_SET);
12.     fwrite(file, phdr.p_filesz, 1, img);
13.     // all zero string bytes
14.     char zero[SECTOR_SIZE];
15.     memset(zero, '\0', SECTOR_SIZE);
16.     // put zero string to the end
17.     fseek(img, SECTOR_SIZE * (*first) + phdr.p_filesz, SEEK_SET);
18.     fwrite(zero, SECTOR_SIZE-(phdr.p_filesz % SECTOR_SIZE), 1, img);
19.     // ceil(phdr.p_memsz)
20.     *nbytes += phdr.p_memsz+SECTOR_SIZE-(phdr.p_memsz%SECTOR_SIZE);
21.     (*first) += (phdr.p_memsz - 1) / SECTOR_SIZE + 1;
22.     // free the buffer
23.     free(file);
24. }
```

注意存储写入的扇区数 first 的初始值为 0 且向上取整，同时，存储写入扇区的字节数 nbytes 计算时要以 SECTOR_SIZE（0x200）为单位向上取整。

为初始化 bss 以及填 0 补齐扇区，直接将 filesz 到扇区结束全部赋 0。

(3) kernel 部分代码

```
1.  void __attribute__((section(".entry_function"))) _start(void){
2.      void (*printstr)(char *) = (void *)0xffffffff8f0d5534;
3.      (*printstr)("\r\nReady for input:");
4.      void (*printchar)(char) = (void *)0xffffffff8f0d5570;
5.      volatile char * state = (char *)0xffffffffbfe00005;
6.      char * input = (char *)0xffffffffbfe00000;
7.      while(1){
8.          if((*state) & 0x01){
9.              (*printchar)(*input);
10.         }
11.     }
12. }
```

结合 C 语言的指针调用函数并查看串口状态，以完成打印与回显任务。

参考文献

[1] The MIPS64 Instruction Set Reference Manual, Revision 6.06

■