Project 1 Bootloader 设计文档

马月骁
2019K8009915025

一、S-core 设计

1. Bootblock 设计

在机器启动时，BIOS 将 SD 卡的第一个扇区拷贝至内存相应的位置。SD 卡中的第一个扇区中为 Bootblock，由于 Bootblock 仅有一个扇区（512B）大小，无法装载整个操作系统，故 Bootblock 的主要任务即是完成操作系统内核从 SD 卡其他扇区到内存的拷贝，随后将控制权移交给 kernel。

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

图 1 Bootloader 拷贝 kernel 代码段

本实验中在拷贝 kernel 时，使用了封装函数“SBI_CALL SBI_SD_READ”。调用该函数时，我们首先需要依据 RISC-V ABI 以及系统调用约定进行传参，随后使用 SBI_CALL 启动系统调用，并提供环境调用号（SBI_SD_READ）。在此过程中，需要对 SBI_CALL 以及 FUNC_NAME（SBI_SD_READ）进行宏替换，将 SBI_CALL FUNC_NAME 转化为对应的系统调用。即首先向 a7 寄存器发送系统调用号，其次执行 ecall 进行系统调用。

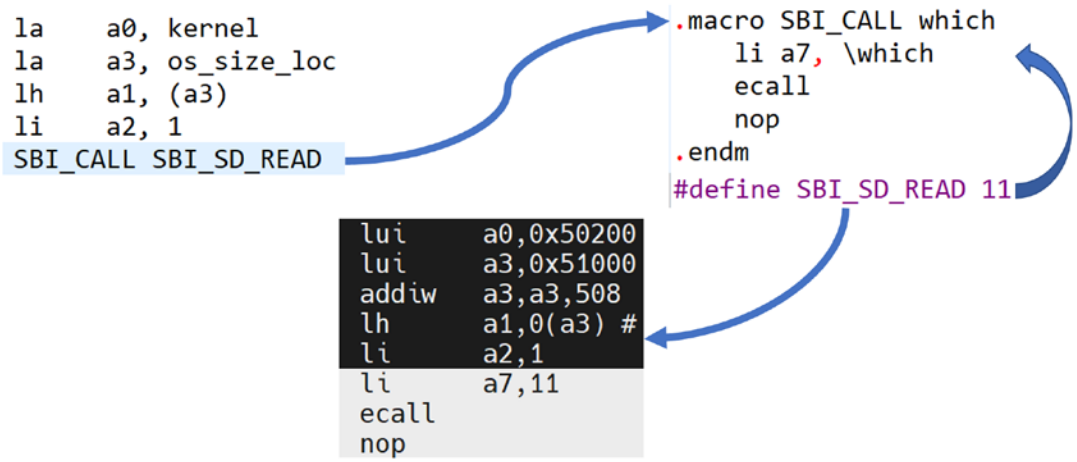

图 2 BIOS 函数调用图解

2. Creatimage 设计

Creatimage 的主要功能为将 bootblock 和 kernel 两个以 ELF 格式存储的二进制文件各段展开，写入 image 镜像中的对应位置，并记录 kernel 占用的扇区数，写入地址 0x502001fc。

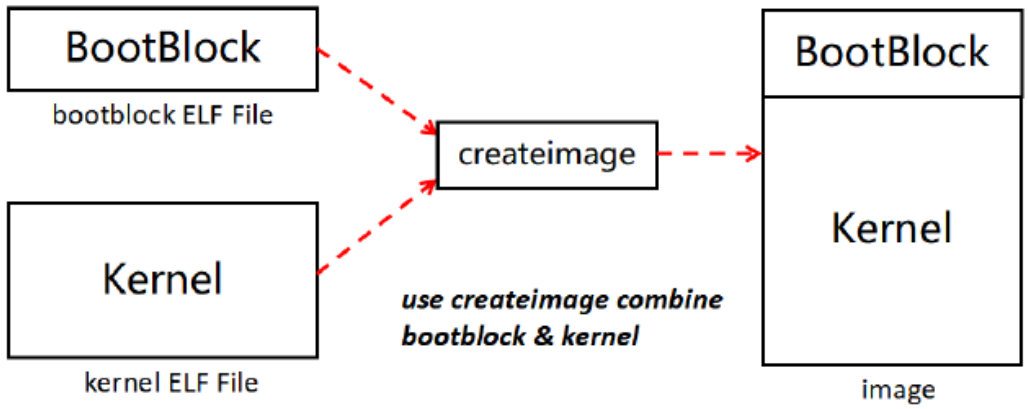

图 3 creatimge 功能图解

对于读取的每个 ELF 文件，creatimage 首先解析 ELF header，获得其程序头数以及各个程序头的偏移量。随后解析各个程序头，读取数据。最后通过 write_segmemt 函数写入 image 镜像。由于每个扇区均为 512B，故对于没有没有占满一个扇区的 segment，我们需要将其剩余空间补 0。

对于 kernel，还需调用 write_os_size 函数，将 kernel 所占用的扇区个数写入 image 镜像（S-core 中地址偏移量为 0x1fc）。

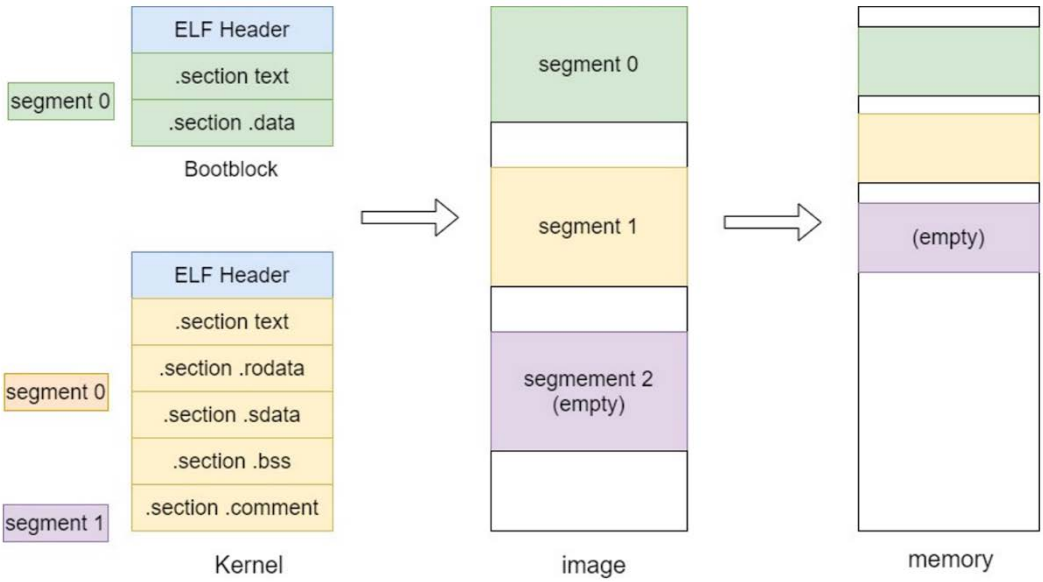

图 4 creatimge 工作过程图解

二、A-core 设计

1. Bootblock 重定位

由于 Bootloader 实际上仅仅在操作系统启动时使用一次，故可将 bootblock 移动至其他地方，将 kernel 加载至 PC 复位的起始地址。在后续运行过程中，bootblock 的内容可以被覆盖，从而节省内存。

为实现此功能，我们需要确定 bootblock 重定位的地址、拷贝 bootblock 以及跳转至重定位后的 bootblock 处三项工作。

（1）Bootblock 重定位地址

bootblock 重定位的地址选择需要满足不会被随后加载的 kernel 直接覆盖导致无法将 kernel 完全拷贝。故可从 0x502001fc 地址处读取出 kernel 所占扇区的大小，进而推算出可以用于重定位的地址范围。在 Project 1 中，由于 kernel 较小，故在实现时采用重定位至固定地址 0x5100000000 的简单实现方法。

（2）拷贝 bootblock

本实验中采用直接将 SD 卡的第一个扇区写入 0x5100000000 处的简单方法。

（3）跳转至重定位后的 bootblock

由于 A-core 中的 bootblock 添加了重定位的功能，若直接跳转至 0x5100000000 处，会出现反复拷贝 bootblock 的死循环。解决方法为跳转时跳过拷贝 bootblock 的代码段，或者在（2）中直接不拷贝拷贝 bootblock 的代码段。在本实验中采用前者。通过 objdump 查看拷贝 bootblock 以及跳转的代码段所占用的空间，跳转时加上对应的偏移量即可。

```
fence
//relocate bootloader
la      a0, bootloader
li      a1, 1
li      a2, 0
SBI_CALL SBI_SD_READ
fence.i
la      t0, bootloader_start
jr      t0
```

图 5 重定位代码段

在执行跳转时，由于此处跳转长度过大，见图 6。而 RISC-V 中 j 是一条伪指令，编译器会用 gp 寄存器做相对寻址，但是这时候还没设置 gp，导致跳转长度不够。解决方案为采用寄存器跳转或是初始化 gp 寄存器。本实验中采取前者。

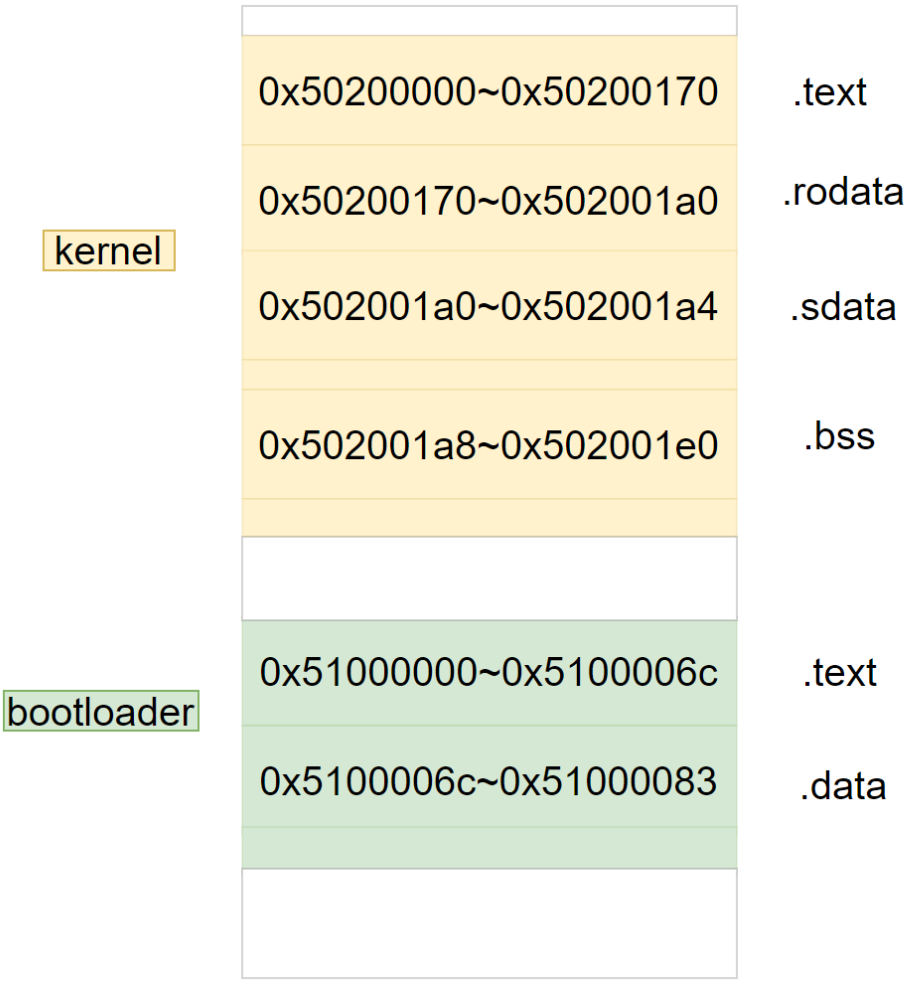

图 6 A-core 中内存分布

2. Creatimage 支持大 kernel 合并

事实上 S-core 中的 creatimage 已经基本可以完成此项功能。其中读取 ELF 文件信息时，对 ELF Header 的解析较为容易，只需全部读出写入 ehdr 即可。而对程序头的解析，其地址需要在程序段偏置（ehdr.e_phoff）的基础上再加一项偏置（ph * ehdr._ephentsize），以确保能读到程序区的每一个段。

三、C-core 设计

C-core 设计要求需要实现双内核的选择驱动。

1. creatimage 设计

事实上，S-core 中的 creatimage 已经基本可以支持此项功能，我们所需增加的功能主要是确定两个 kernel 所占扇区数写入的地址。

```
#define OS_SIZE_LOC 0x1f8
static void write_os_size(int nbytes, FILE *img, int file_num)
{
    int os_size = nbytes / 512;
    fseek(img, (long)(OS_SIZE_LOC + (file_num - 1) * 4), SEEK_SET);
    fwrite(&os_size, 2, 1, img);
    printf("os_size: %d sectors\n", os_size);
}
```

图 7 kernel 大小写入

如图 7 所示，两 kernel 大小写入地址的偏移量分别为 0x1f8 与 0x1fc。

2. Bootblock 设计

Bootblock 所需要添加的功能是实现通过读取输入，确定启动哪一个 kernel。

通过调用 SBI_CONSOLE_GETCHAR 函数来实现读取输入。

```
        //task4 choose kernel
        li      t0, -1
L0:
        SBI_CALL SBI_CONSOLE_GETCHAR
        fence.i
        //SBI_CALL SBI_CONSOLE_PUTCHAR
        beq a0, t0, L0
```

图 8 读取输入代码

依据输入字符的大小，决定拷贝 kernel_1 或 kernel_2。代码见图 9。

Kernel_1 的起始扇区号为 1，大小存在 0x510001f8；在加载 kernel_2 时，可以通过读取 kernel_1 的大小进而计算出 kernel_2 的起始扇区号，其大小存储在 0x510001fc。

```
        lui     t0, %hi(os_size_loc)
        addi    t0, t0, %lo(os_size_loc)
        li      a1, 49
        beq a0, a1, L1

        //load kernel_2
        lh      a1, 4(t0)
        lh      a2, (t0)
        addi    a2, a2, 1
        j       L2

L1:
        //load kernel_1
        lh      a1, (t0)
        li      a2, 1

L2:
        // 2) task2 call BIOS read kernel in SD card and jump to kernel start
        la              a0, kernel                                      //mem_address
        //la            a3, os_size_loc
        //lh            a1, (a3)                                        //num_of_blocks
        //li            a2, 1                                           //block_id
        SBI_CALL SBI_SD_READ
        fence.i
        la              t0, kernel_main
        jr              t0
```

图 9 kernel 选择代码

修改好 Makefile 文件后重新编译，运行效果如下：

```
bbl(NutShell) > loadbootstart sd load
SDIO operation: 0
Base sector: 71680
# of sectors: 1
Memory address: 50200000
0000000000000800
SDIO operation: 0
Base sector: 71680
# of sectors: 1
Memory address: 51000000

It's a bootloader...
Type 1 to choose kernel_1, else kernel_2
SDIO operation: 0
Base sector: 71681
# of sectors: 2
Memory address: 50200000
Hello OS! This is kernel_1!
bss check: t version: 1
```

```
bbl(NutShell) > loadbootstart sd load
SDIO operation: 0
Base sector: 71680
# of sectors: 1
Memory address: 50200000
0000000000000800
SDIO operation: 0
Base sector: 71680
# of sectors: 1
Memory address: 51000000

It's a bootloader...
Type 1 to choose kernel_1, else kernel_2
SDIO operation: 0
Base sector: 71683
# of sectors: 2
Memory address: 50200000
Hello OS! This is kernel_2!
bss check: t version: 1
```