Project1 Bootloader 设计文档

中国科学院大学
[薛峰]
[2018/9/25]

1. Bootblock 设计流程

（1）Bootblock 主要完成的功能

Bootblock 通过 BIOS 调用读取 SD 卡/硬盘上的操作系统内核，将其存放到内存中的指定位置。随后跳转到操作系统的入口代码，使操作系统开始运行。

（2）Bootblock 被载入内存后的执行流程

将操作系统在内存中的起始地址、操作系统内核在 SD 卡上的偏移量以及操作系统内核的大小传给 read_sd_card 函数，使得操作系统内核能够被读入内存中。随后跳转到内核入口地址，开始执行操作系统内核。

（3）Bootblock 如何调用 SD 卡读取函数

分别将操作系统系统在内存中的起始地址 addr、操作系统内核在 SD 卡上的偏移量 offset 以及操作系统内核的大小 size 传给 SD 卡读取函数(即分别传给寄存器 a0,a1,a2)，随后跳转到 SD 卡读取函数的入口地址。

（4）Bootblock 如何跳转至 kernel 入口

kernel 入口是固定的(假设记为 0x80000200)，当 bootblock 将内核读入内存后，通过指令 jal 0x80000200 便可跳转至 kernel 入口。

（5）任何在设计、开发和调试 bootblock 时遇到的问题和解决方法

a)问题：不知道如何让 kernel.c 中打印出 Hello OS!

解决方法：上网查之后发现#define 中的 xbfe48000 是 UART 控制器 EX FIFO 的地址。想起来上学期组成原理实验课使用过这个控制器。于是查看上学期讲义并解决了该问题。

b)问题：只打印出 It’Boot Loader!而不打印 Hello OS!

解决方法：最开始认为 SD 卡出了问题，不能传入数据。当换用同学的 SD 卡后发现问题仍没有解决。后仔细检查代码，发现 bootblock.s 中代码如下：

```
li  $a0, 0xa0800200
li  $a1, 0x512
li  $a2, 0x512
jal 0x8007b1cc      #read_sd_card
```

其中我将操作系统在 SD 卡上的偏移量 offset 赋了 0x512，而不是 0x200。因此 bootlock 载入的是无效的指令，所以不会打印出 Hello OS!

Createimage 设计流程

（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

createimage 将 bootblock(bootblock.s 编译后的 elf 文件)和 kernel(kernel.s 编译后的 elf 文件)中的各个段写入 image 中。其中 bootblock 在前 512 字节，kernel 在之后的

位置。

（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

根据其 elf 文件的文件头可以找到其程序头的偏移量和程序头的个数。因为其程序头在连续的位置存放，因此根据偏移量和个数可以得到所有的程序头。各个程序头中可以找到对应 segment 的偏移量和大小。根据偏移量可以找到可执行代码位置。将所有的 segment 的大小相加可以得到所有可执行代码的大小。

（3）如何让 Bootblock 获取到 Kernel 二进制文件的大小，以便进行读取

createimage 将 kernel 二进制文件的大小存放在 bootblock 所在扇区的最后一个字节即第 511 字节。bootblock 只需要用 load 指令把该位置存放的数取出，便可以知道 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

a) 问题：一开始不知道如何读文件写文件，对项目的思路很清晰但不知道如何写代码。

解决方法：询问同学之后，他们告诉我可以用 fopen，fread 等函数，随后上网查这些函数的使用方法，从而顺利完成实验。

b)问题：对于如何让 bootblock 获取 kernel 大小的这个问题，最开始想的是通过反汇编，找到 bootblock 中调用 read_sd_card 函数所在的位置，将其中的参数改为 kernel 的大小。

解决方法：在 design review 的时候，蒋德钧老师提供了一个新的思路，在蒋老师的提示下，我想到了吧 kernel 的大小放入 bootblock 所在扇区的最后一个字节，bootblock 只需要用一条 load 指令便可取出这个数。

c)问题：对于 count_kernel_sectors 指令，需要知道程序头的个数，但是传入的参数只有一个程序头的指针。

解决办法：在结尾增添一个标志位，即多加一个程序头，将该程序头的 e_flag 置为-1，通过这个办法便可以知道程序头的个数。

2. 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）

（1）bootblock 中调用调用 read_sd_card 代码部分：

```
li   $t0, 0xa0800000

li   $a0, 0xa0800200
li   $a1, 0x200
lw   $a2, 0x1fc($t0)
jal 0x8007b1cc       #read_sd_card
```

首先将 0xa0800000 给寄存器 t0 作为一个基址，后将 addr 和 offset 参数传入，通过 lw 指令取出 0xa08001fc 位置上仿制的 kernel 的大小，赋给 a2，至此所有参数均已传给 read_sd_card 函数。

（2）kernel.c 中打印字符串代码部分：

```c
#define PORT 0xbfe48000

void __attribute__((section(".entry_function"))) _start(void)
{
  int i;
  char s[] = "Hello OS!";

  for(i = 0; i < 9; i++)
    *(char*)PORT = *(s+i);

  return;
}
```

每次向端口传入 1 个 char 型数据，这样便可打印出该字符串。

（3）read_exec_file 函数代码：

```c
Elf32_Phdr *read_exec_file(FILE *opfile)
{
  FILE * file = opfile; //##
  Elf32_Ehdr * Ehdr;
  Elf32_Phdr * Phdr;

  if( (Ehdr= (Elf32_Ehdr *) malloc(52)) == NULL)  //malloc Ehdr
  {
    printf("malloc Ehdr failed!\n");
    exit(-1);
  }

  fread(Ehdr, 52, 1, file);              //get the ELF header

  fseek(file, Ehdr->e_phoff, SEEK_SET);  //seek the first program header
  if( (Phdr = (Elf32_Phdr *) malloc( ((Ehdr)->e_phnum + 1)*sizeof(Elf32_Phdr) )) == NULL)  //malloc Phdr
  {
    printf("malloc Phdr failed!\n");
    exit(-1);
  }
  fread(Phdr, sizeof(Elf32_Phdr), Ehdr->e_phnum, file);  //get all program header
  Phdr[Ehdr->e_phnum] = Phdr[Ehdr->e_phnum - 1];
  Phdr[Ehdr->e_phnum].p_flags = -1;      //identify the end

  free(Ehdr);
  Ehdr = NULL;

  return Phdr;
}
```

主要思想为通过文件头找到程序头的 offset 和程序头的个数。从而可以定位到各个程序头，将这些程序头放在一个数组中。在数组的末尾加入一个标志位，从而使得其他函数根据该标志位可以知道程序头的个数。

（4）record_kernel_sectors 函数：

```c
void record_kernel_sectors(FILE *image, int kernelsz)
{
  fseek(image, 508, SEEK_SET);
  fwrite(&kernelsz, 4, 1, image);
}
```

将 kernelsz 存入 bootblock 所在扇区的最后一个字节。其目的是使 bootblock 取该位置的数据从而得到 kernel 的大小。

（5）write_kernel 函数代码：

```
void write_kernel(FILE *image, FILE *knfile, Elf32_Phdr *Phdr, int kernelsz)
{
  char buffer[kernelsz*512];
  int i;
  int cur_capacity = 0;
  for(i = 0; Phdr[i].p_flags != -1; i++)
  {
    fseek(knfile, Phdr[i].p_offset, SEEK_SET);
    fread(buffer + cur_capacity, Phdr[i].p_filesz, 1, knfile);
    cur_capacity += Phdr[i].p_filesz;
  }
  fwrite(buffer, cur_capacity, 1, image);
}
```

主要思想是首先建立一个 buffer 的空间，用于临时存放从 kernel 的 elf 文件取出来的可执行代码。其中可执行代码的定位和大小已在“Createimage 设计流程”的问题(2)中阐述。之后，用 fwrite 函数将 buffer 内的内容写入 image 中即可。

（6）Bonus 部分代码

```
main:
  la  $a0, msg
  jal 0x8007b980      #printstr

  li  $a0, 0xa0800000
  li  $a1, 0x200
  lw  $a2, 0xa08001fc($0)

  li  $31, 0xa0800000
  j 0x8007b1cc       #read_sd_card
```

主要思想为对跳转到 read_sd_card 的 jal 指令进行修改。给 a0 寄存器赋 0xa0800000，即将 bootload 覆盖，给 31 号寄存器赋值为 0xa0800000，使用 J 指令跳到 read_sd_card 函数，之后 read_sd_card 函数执行过后便可以跳到 0xa0800000 即 kernel 的入口地址。通过上述方法便可将 bootload 覆盖。

■