# Project 1 Bootloader 设计文档

中国科学院大学 徐泽凡 2018K8009929037

## Bootblock 设计

### (1) Bootblock 的主要功能

Bootblock 主要完成的功能如下：

- 从 SD 卡中重新复制一份 bootblock 到内存中；
- 输出 `It's a bootloader...`
- 从 SD 卡中复制 kernel 到 `0xffffffffa0800000`
- 跳转到 kernel 继续运行

### (2) Bootblock 调用 SD 卡读取函数的方法

基本思路：

- 将参数 `dest`、`offset`、`size` 装载至寄存器 `$a0`、`$a1`、`$a2`。
- 将函数入口地址装载至某一临时寄存器（如 `$t0`）。
- 通过 `jalr` 指令跳转至前面临时寄存器（`$t0`）所保存的地址。

参考代码：

```
#define BOOT_NEW_LOC    0xffffffffa0900000
#define BOOT_OFS        0x200
#define SECTOR_SZ       0x200
#define READ_SD_CARD    0xffffffff8f0d5e10

dli     $a0, BOOT_NEW_LOC       # dest:     new address of bootblock
dli     $a1, BOOT_OFS           # offset:   bootblock offset in sd card
dli     $a2, SECTOR_SZ          # size:     one sector
dli     $t0, READ_SD_CARD       # read_sd_card(dest, offset, size)
jalr    $t0
```

### (3) Bootblock 跳转至 kernel 入口的方法

基本思路：

- 将 kernel 入口地址装载到某一临时寄存器（如 `$t0`）。
- 通过 `jr` 指令跳转至前面临时寄存器（`$t0`）所保存的地址。

参考代码：

```
dli     $t0, KERNEL_ADDR        # jump to kernel
jr      $t0
```

### (4) 其他问题

为了简化汇编编程，汇编器提供了一系列不在指令集中的指令、指令语法，它们提供了一些较复杂的常用功能，通常在编译时会被汇编器翻译成数条机器指令。

以 `dli` 装载 64 位立即数举例，下面的这条指令：

```
dli     $t0, 0xffffffff8f0d5e10
```

会被汇编器翻译为：

```
lui     t0, 0x8f0d
ori     t0, t0, 0x5e10
```

类似的还有，`ld` 加载双字（可以自动装载标号地址后再进行寄存器间接寻址）、`dla` 加载 64 位地址等指令。这样的设计简化了 MIPS 这类精简指令集的汇编语言中一些繁琐的细节，降低了编程难度。

原先的代码框架中，几个函数的入口地址、kernel 的入口地址都是以数据的方式存储在 `.data` 段中的。我不太清楚这样做有什么优势，在后续重定位的实验中，我将之改为宏定义和加载立即数的操作。

# Createimage 设计

## (1) createimage 涉及到的二进制文件

createimage 程序需要处理这 3 种二进制文件：Bootblock 编译后的二进制文件、Kernel 编译后的二进制文件、写入SD卡的 image 文件。

image 镜像文件，仅包含了 Bootblock、Kernel 编译后的二进制文件的程序段（segment）部分，这部分是实际的可执行代码和数据。createimage 负责从 Bootblock、Kernel 编译后的二进制文件抽取程序段，并将它们按扇区放置到 image 中。

image 的结构如下：

```
File "image"
0x000:  Program segment of Bootblock
0x200:  Program segment of Kernel
```

## (2) 获取二进制文件中的可执行代码的位置和大小

Bootblock、Kernel 编译后的二进制文件符合 ELF 格式，createimage 在打开它们后依次进行下面的操作：

- 从文件头部读取 ELF 文件头；
- 根据 ELF 文件头中的 `e_phoff` 和 `e_phentsize` 确定程序头的位置，并从文件中读取 ELF 程序头。
- 根据 ELF 程序头中的 `p_offset`、`p_memsz` 和 `p_filesz` 确定程序段（segment）的位置。

ELF 文件头、ELF 程序头分别符合 `elf.h` 中定义的 `Elf64_Ehdr`、`Elf64_Phdr` 结构体，可以直接从文件中读取到声明的结构体中。示例代码如下：

```
Elf64_Ehdr ehdr;
fread(ehdr, 1, sizeof(Elf64_Ehdr), fp);
Elf64_Phdr phdr;
fread(phdr, 1, sizeof(Elf64_Phdr), fp);
```

实际开发中，Kernel 只包含 1 个 segment，只从 kernel 的可执行代码拷贝了 1 个 segment。

## (3) 获取到 Kernel 的大小

createimage 文件在处理 Kernel 的二进制文件时，很容易获取 kernel 的大小。由于 Bootblock 不可能占满一个扇区，我们可以把 kernel 的大小放在扇区的最后两个字节中，即 `0x1fe` 处。由于 Kernel 部分的大小是扇区

的整倍数，我们只需要存储扇区数即可。相关代码如下：

```
static void write_os_size(int nbytes, FILE *img) {
    uint16_t os_size = nbytes / 0x200;
    printf("os_size: %d sectors\n", os_size);
    fseek(img, OS_SIZE_LOC, SEEK_SET);
    fwrite(&os_size, sizeof(uint16_t), 1, img);
}
```

对于 Bootloader，在从 SD 卡复制 kernel 前，需要先从 `bootblock 入口地址 + 0x1fe` 处加载 kernel 的扇区数。而后乘以 `0x200` 得到字节数。相关代码如下：

```
dla     $a2, main + OS_SZ_OFS
lh      $a2, ($a2)
dmul    $a2, $a2, 0x200
```

### 其他问题

MIPS 实验框架中的 `createimage.c` 内容较 RISC-V 缺少了很多，实验中我在补全基础框架（如处理命令行选项）花了相当长时间。但在这个过程中，我也学习到了一些用于处理命令行选项的库函数、处理可变参数的宏等等。

## A-Core/C-Core 设计

要实现重定位，需要保证向内存中复制 kernel 的时候，bootblock 的运行不能中断。一个简单的想法是重新复制一份 bootblock 至 kernel 后面的内存空间，然后跳转至新的 bootblock 执行加载 kernel 的工作。

这里有不少细节需要处理：

- 将第二份 bootblock 复制到哪里：复制到一个固定的地方，我选择的是 `0xffffffffa0900000`。
- bootblock 的入口地址如何设置：在 `Makefile` 中，修改入口地址到第二份 bootblock 的预期位置。
- 怎么复制：通过调用 `read_sd_card` 函数，复制第一个扇区到 `0xffffffffa0900000`，需要注意这一段复制代码必须是地址无关的。
- 怎么跳转：在复制第二份 bootblock 后，添加标号 go，通过 `j` 指令跳转到 go。
- kernel 的入口地址：需要修改为 `0xffffffffa0800000`。

在完成第二份 bootblock 复制并跳转后，bootloader 的运行不再有任何限制。如果 Bootloader 在加载 Kernel 后还有其他工作要完成，可以正常工作。

这一部分代码请参见后文"关键函数功能"一节。

### 其他问题

在最初考虑重定位问题时，我试图让第二份 bootblock 的位置随着 Kernel 的大小而改变。但随后发现这是基本不可能的，各类标号的值都是与地址相关的，可变的位置会带来各种问题。

在开发过程中，bootloader 的入口地址是第一座山。在编译过程中，我们为程序指定了一个预期的入口地址，但实际上程序会被复制到哪里，是由 BIOS（复制 Bootblock）和 Bootloader（复制 kernel）决定的。这个预期的入口地址，决定了汇编程序中各类标号的实际地址。为了完成重定位的任务，我们需要想办法让 bootblock 在两个不同的地址空间运行，这里有两种方法：

- 通过修改链接脚本，让复制前后两个部分分别有不同的预期入口地址。
- 整个 bootloader 只有一个确定的入口地址，但通过设计使得复制前（或复制后）运行的部分是地址无关的。

我研究了很久链接器脚本，也没有研究出来如何做到第一种方式。第二种方式相对简单，复制前只需要执行复制第二份 bootblock 的工作，这一部分很容易做到与地址无关，bootloader 的入口地址设置为复制后的地址即可。

第二座山，是如何跳转到复制后的 bootloader，跳过复制 bootloader 继续执行。最初考虑的时候，我试图获取 PC 的值，然后通过两份 bootblock 的距离计算跳转到的地址，但并没有找到任何能够合理地获取 PC 的方法。后来研究标号的机制后发现，如果指定 bootloader 的入口地址为第二份 bootblock 的地址，那么可以直接在后半部分程序的开头处添加标号，通过 `j label` 这样的指令跳转到第二份 bootblock 的位置继续执行。也可以先 `dla` 加载地址再 `jr` 跳转到寄存器实现，汇编器会将 `label` 替换为根据入口地址计算的绝对地址。

除了上述的重定位方法，在其他同学的提示下，我还实现了另一种"剑走偏锋"的方法。在复制 kernel 时，计算机离开 bootblock 跳转到 `read_sd_card` 函数执行，在复制完成后又回到 bootblock 调用函数的地方继续执行。如果 `read_sd_card` 函数不再返回到 bootblock 而是直接进入 kernel 执行，就可以"巧妙地"解决覆盖地问题。具体实现思路如下：

- 修改寄存器 `$ra` 的值为 kernel 的入口地址；
- 改用 `jr` 而不是 `jalr` 跳转到 `read_sd_card` 函数。

但是，这种做法是不符合 ABI 约定的，不推荐使用。

# 关键函数功能

## bootblock 的重定位功能代码

此部分代码源自 `bootblokc.S`，负责完成拷贝第二份 bootblock 的工作并跳转执行。

```
main:
# copy bootblock to new place (This part is position-independent)
    dla     $a0, main               # dest:     new address of bootblock
    dli     $a1, BOOT_OFS           # offset:   bootblock offset in sd card
    dli     $a2, SECTOR_SZ          # size:     one sector
    dli     $t0, READ_SD_CARD       # read_sd_card(dest, offset, size)
    jalr    $t0

    j       go                      # jump to new bootblock

go:
    # subsequent code...
```

这里需要注意 `dla $a0, main` 一行，通过 `main` 标号加载入口地址，可以避免入口地址同时出现在 `Makefile` 和 `bootblock` 中出现，方便修改。

## createimage 读取 ELF 文件头和程序头、写入 segment

此部分代码源自 `createimage.c`，负责读取 ELF 文件头和程序头、写入 segment。

```
static void read_ehdr(Elf64_Ehdr *ehdr, FILE *fp) {
    fseek(fp, 0, SEEK_SET);
    int nbytes = fread(ehdr, 1, sizeof(Elf64_Ehdr), fp);
    if (nbytes != sizeof(Elf64_Ehdr)) {
        error("ERROR: Cannot read ELF header.\n");
    }
}

static void read_phdr(Elf64_Phdr *phdr, FILE *fp, int ph, Elf64_Ehdr ehdr) {
    if (ph >= ehdr.e_phnum) {
```

```
        error("ERROR: Segment %d not found, only have %d segments.\n", ph, ehdr.e_phnu
    }
    fseek(fp, ehdr.e_phoff, SEEK_SET);
    fseek(fp, ph * ehdr.e_phentsize, SEEK_CUR);
    int nbytes = fread(phdr, 1, sizeof(Elf64_Phdr), fp);
    if (nbytes != sizeof(Elf64_Phdr)) {
        error("ERROR: Cannot read program header.\n");
    }
}


static void write_segment(Elf64_Ehdr ehdr, Elf64_Phdr phdr, FILE *fp, FILE *img, int *
    fseek(fp, phdr.p_offset, SEEK_SET);
    fseek(img, *total_nbytes, SEEK_SET);

    unsigned char * buff = malloc(phdr.p_memsz);
    memset(buff, 0, phdr.p_memsz);

    int nbytes = fread(buff, 1, phdr.p_filesz, fp);
    if (nbytes != phdr.p_filesz) {
        error("ERROR: Cannot read segment\n");
    }

    nbytes = fwrite(buff, 1, phdr.p_memsz, img);
    if (nbytes != phdr.p_memsz) {
        error("ERROR: Cannot write segment\n");
    }

    *total_nbytes += nbytes;
    free(buff);

    if (options.extended) {
        printf("\t\twriting 0x%04x bytes\n", nbytes);
    }
}
```

代码中有相当部分是在进行错误处理，核心部分内容并不多。

### kernel 回显代码

此部分代码源自 `kernel.c`，负责实现回显功能。这部分代码简单的完成了一个类似于命令提示符的显示，并处理了换行的问题。

```
#define FUNCTION_PRINTSTR   0xffffffff8f0d5534
#define FUNCTION_PRINTCH    0xffffffff8f0d5570
#define COM_STATUS_REG      0xffffffffbfe00005
#define COM_DATA_REG        0xffffffffbfe00000

void (*printstr)(char * str) = (void *) FUNCTION_PRINTSTR;
void (*printch)(char ch) = (void *) FUNCTION_PRINTCH;
volatile char * com_status_reg = (void *) COM_STATUS_REG;
volatile char * com_data_reg = (void *) COM_DATA_REG;


void __attribute__((section(".entry_function"))) _start(void)
```

```
{
    printstr("Hello OS\r\n");
    char com_data;
    printstr("$ ");
    while (1) {
        if ((*com_status_reg) & 0x1) {
            printch(com_data = *com_data_reg);
            if (com_data == '\r') {
                printstr("\n$ ");
            }
        }
    }
}
```

## 致谢

感谢在 Design Review 时与我交流的蒋德钧老师，与他的交流让我受益匪浅。

感谢我的队友徐幡同学，以及在实验过程中与我交流的其他各位同学。