Project1 Bootloader 设计文档

中国科学院大学
李昊宸
2017K8009929044

BOOTLOADER 设计流程

（1） BOOTBLOCK 完成的功能

在任务一中，Bootblock 作为系统启动时运行的第一个文件，其作用为调用 BIOS 函数 printstr（char *string）打印字符串“It‘s a bootloader..”
在任务二中，Bootblock 将 kernel 加载到内存中，并跳转到 kernel 执行。

（2） BOOTBLOCK 的执行过程

对于龙芯处理器而言，在 CPU 上电后，执行地址会在 PMON 的调控下自动跳转至一处可执行代码，这段代码的作用是把 sd 卡上第一个扇区（512B）的内容拷贝到开发板上 0xa0800000 的位置。Bootblock 在烧写后被放置在 sd 卡的第一个扇区 sdb，被连接在了可执行文件的开头位置，插入 sd 卡后将自动加载至内存。随后 pc 从入口地址 0xa0800000 开始执行 Bootblock。

① 打印字符串

通过 la 指令将要打印的字符串地址送至传参寄存器$a0，然后通过 jal 指令跳转值 printstr 函数的地址 0x80011100 执行打印字符串

② 调用 SD 卡读取函数

调用完 printstr 函数，pc 会在 jal 的引导下返回 bootblock 执行下一行指令。在这里我们需要调用 read_sd_card 函数将 kernel 写入内存。所调用的 *BIOS 函数 read_sd_card(addr, offset, size)有如下三个参数：*

addr	*移动到内存的目标位置 0xa0800200*	*存至$a0*
offset	*要移动数据在 SD 卡中的偏移量 Bootblock 占据第一个扇区，Kernel 在第二个扇区，偏移量为 0x200*	*存至$a1*
size	*Kernel 的大小为一个扇区，占 512B，为 0x200 字节*	*存至$a2*

随后用 jal 跳转到 0x80011000 地址处的 read_sd_card 函数开始执行拷贝。
另外，为什么我们要把 Kernel 放到 0xa0800200 这里呢？一个原因是因为 512B 大小的 bootblock 占据了前面的分区，地址顺延。

③ 跳转至 kernel 入口

我们需要查看 ld.script 是怎样完成链接工作的。
右图是链接器的部分段落，我么可以看到在 SECTIONS 分区下的第一个节.text 中，被第一个链接的文件是入口函数 entry_function，于是 Kernel 的入口地址应该为 Kernel 被拷贝到的首地址 0xa0800200，也就是 Kernel_main 的地址。采用 jal 指令跳转至该地址执行即可。

```
SECTIONS
{
  . = 0xffffffff80000000;
  .exc_handler : {
  *(.exception_handler)
  }
  . = 0xffffffffa0800000;
  .text :
  {
    _ftext = . ;
    *(.entry_function)
    *(.text)
    *(.rodata)
    *(.rodata1)
    *(.reginfo)
    *(.init)
    *(.stub)

    *(.gnu.warning)
  } =0
```

(3) 遇到的问题和解决方法

① 起初在调用 printstr 函数时，直接使用语句 jal printstr 会导致问题的产生。因为在.s 文件中，printstr 作为一个标号，jal 语句会使 pc 跳转值该行继续执行。但是此处 printstr 被定义为一个地址，实际的 BIOS 程序并不在这里存放，于是导致问题产生。应该改用 jal 0x80011100

② 在向寄存器引入参数时，起初使用了 la $a0, kernel 指令，但在运行过程中发现并没有产生正确结果。原因出现在 kernel 作为标号时，其本身不作为一段有实义的文本，而是作为一段地址。我们考虑以下例子：

```
.section .data          output: .ascii "sum = "
......
la      $a1, output
```

在这里，output 作为一个标号，承载着的是后文 ascii 码的首字符，其地址对应的就是“s”所存储的地址。

```
kernel : .word 0xa0800200
```

而 kernel 被定义为一个 word 型文本，其地址指向的是存放文本 0xa0800200 的首地址，假设记该地址为 b，那么最终传入$a0 的是 b 而不是我们想要的 0xa0800200,故出现错误。

③ C 语言对函数的调用，可由函数指针语句实现。

1) 在全局区域声明 void （*函数名）(参数类型 参数名)

2) 在 main 中声明 函数名 = (void) 函数地址

3) 调用 (*函数名)(参数)

(4) 实现代码段

① bootblock.c main 函数段

```
.text
.global main

main:
        # 1) task1 call BIOS print string "It's bootblock!"
        la   $a0, msg
        jal  0x80011100
        # 2) task2 call BIOS read kernel in SD card and jump to kernel start
        la   $a0, 0xa0800200
        li   $a1, 0x200
        li   $a2, 0x200

        jal  0x80011000
        jal  0xa0800200
# while(1) --> stop here
stop:
        j stop
```

② kernel.c

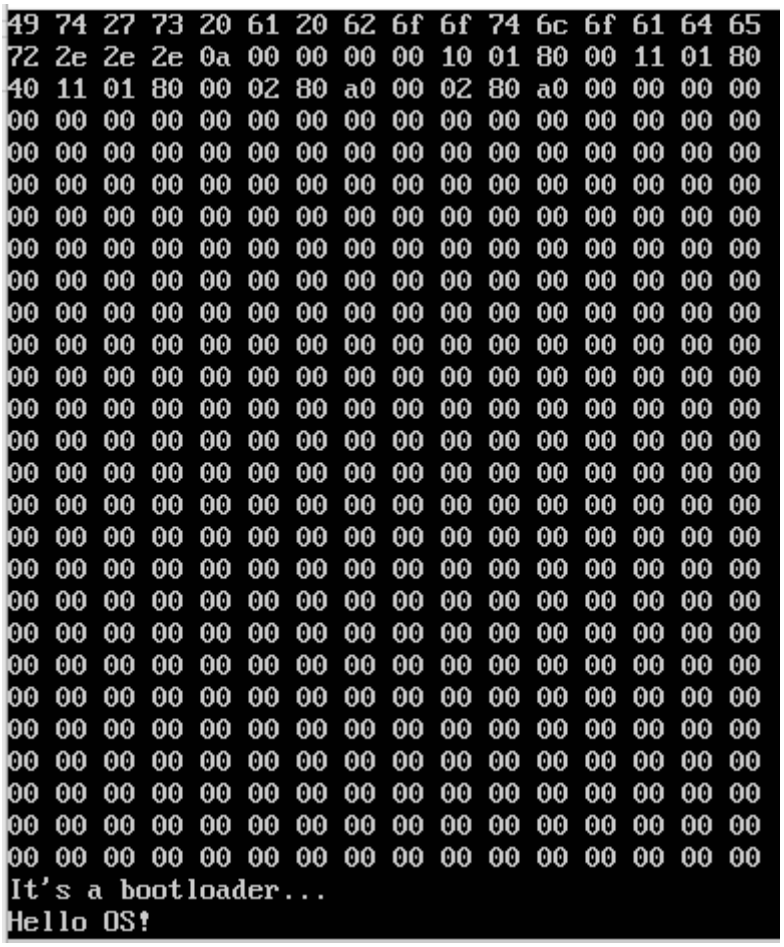
```
void (*printstr)(char *str);

void __attribute__((section(".entry_function"))) _start(void)
{
        // Call PMON BIOS printstr to print message "Hello OS!"

        char message[] = "Hello OS!";
        printstr = (void *)0x80011100;
        (*printstr)(message);

        return;
}
```

（5） 运行结果


49 74 27 73 20 61 20 62 6f 6f 74 6c 6f 61 64 65
72 2e 2e 2e 0a 00 00 00 00 10 01 80 00 11 01 80
40 11 01 80 00 02 80 a0 00 02 80 a0 00 00 00 00
It's a bootloader...
Hello OS!

(1) BOOTBLOCK BONUS 完成的功能

实现覆盖 bootblock 写入，即将 kernel 写入内存的地址改为 Bootblock 起始地址。

(2) BOOTBLOCK 的执行过程

由于覆盖之后，从 SD 卡读取函数返回时，返回地址需要变为 0xa0800000，因此，手动写入 ra，跳到 SD 卡读取函数执行。

(3) 遇到的问题和解决方法

① 在只修改 bootblock 后，发现了问题。定义在 kernel 程序内部的局部变量 char *str = “Hello OS!\n”被打印出了乱码。查阅 Makefile 后找到了这样的语句：

```
bootblock: bootblock.s

	${CC} -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=0xffffffffa0800000 -N -o bootblock bootblock.s -nostdlib -e main -Wl,-m -Wl,elf32ltsmip -T ld.script

kernel: kernel.c

	${CC} -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=0xffffffffa0800200 -N -o kernel kernel.c -nostdlib -Wl,-m -Wl,elf32ltsmip -T ld.script
```

不难发现生成 image 二进制文件时，地址顺序已被编译器安排好，kernel 在 bootblock 之后的 0x200 偏移量处。在转入内存时，我们所人工覆盖掉的 bootblock 相当于将整体向前移动 0x200 个字节，所以在打印时地址指针参数需要减 0x200。

② 由于修改了$ra 寄存器，kernel 函数在返回 return 时不断的循环执行。为了终止循环打印，添加 while（1）{}来使程序停止。

(4) 实现代码段

① bootblock.c main 函数段

```
.text
.global main

main:
        # 1) task1 call BIOS print string "It's bootblock!"
        la   $a0, msg
        jal  0x80011100
        # 2) task2 call BIOS read kernel in SD card and jump to kernel start
        la   $a0, 0xa0800000      #kernel覆盖bootblock
        li   $a1, 0x200
        li   $a2, 0x200
        li   $ra, 0xa0800000      #手动存储返回地址    如此一来在函数执行结束之后返回的就是kernel的开头

        j    0x80011000           #执行read_sd_Card

# while(1) --> stop here
stop:
        j stop
```

② kernel.c

```
void (*printstr)(char *str);
//char  *value = "2019\n\n";
void __attribute__((section(".entry_function"))) _start(void)
{
        // Call PMON BIOS printstr to print message "Hello OS!"


        char *message = "Hello OS!\n";
        char *v = "Version: 2019\n\n";
        char *signature = "made by moto!";
        printstr = (void *)0x80011100;
        (*printstr)(message-0x200);
        (*printstr)(v-0x200);

        (*printstr)(signature-0x200);
        while(1){}


        return;
}
```

(5) 运行结果

```
28 04 00 00 c1 00 00 00 00 00 00 00 00 00 00 01
It's a bootloader...
Hello OS!
Version: 2019

made by moto!
```

CREATEIMAGE 设计流程

（1） CREATEIMAGE 完成的功能

Createimage 的功能主要是将 bootblock 二进制文件和 kernel 二进制文件编译成为一个可以放入芯片内存执行的 image 文件，也就是我们所将要制作的内存镜像。

image 文件由两部分组成：

第一部分是针对交叉编译形成的 bootblock 二进制文件，去掉了文件头(ELF header)和程序头（Program segment header）后，剩下的 segment 段（由多个 section 组成），随后会被放置于 SD 卡中的第一扇区；

第二部分是针对交叉编译形成的 kernel 二进制文件，去掉了文件头（ELF header）和程序头（Program segment header）后，剩下的 segment 段（由多个 section 组成），随后会被放置于在 SD 卡中的第二扇区。

（2） CREATEIMAGE 的执行过程

Createimage 需要实现多个功能，单独解释过于复杂，直接结合代码进行叙述：

① 读取 ELF 文件中的程序头（Program segment header）

```
Elf32_Phdr *read_exec_file(FILE *opfile)              //读取elf格式的文件中的程序头
{
    Elf32_Phdr *phdr;
    Elf32_Ehdr *ehdr;
    uint8_t buffer[512];                              //分配一个扇区大小的缓冲区
    phdr = (Elf32_Phdr *)calloc(1,sizeof(Elf32_Phdr));   //分配一个程序头大小的区域，并将数据初始化为0
    fread(buffer, 1, 512, opfile);      //size_t   fread(void *buffer, size_t   size, size_t  count, FILE  *stream );读取512个字节到buffer
    ehdr = (Elf32_Ehdr *)buffer;                      //把缓冲区文件转换为文件头的格式，以读取程序头的偏移量
    memcpy(phdr,(void *)ehdr + ehdr->e_phoff,32 );      //程序头的大小为32个字节       e_phoff为程序头到文件首地址的偏移量
    return phdr;

}
```

该函数实现的功能为返回一个文件中的程序头。首先初始化一个程序头 Elf32_Phdr 格式的变量 phdr，声明一个文件头 Elf32_Ehdr 类型的变量 ehdr。用 fread 函数将 file 读取至缓冲区 buffer 中，再通过强制类型转换把 buffer 中的数据提取到 ehdr 中，并转为文件头类型，从而将文件头中程序头的偏移量读取出来。最后，直接在文件中对应的位置中 copy 32 字节大小的内容（因为程序头的大小为 32 字节），并将其返回。

② 计算 kernel 的 segment 段有多少扇区

```
uint8_t count_kernel_sectors(Elf32_Phdr *Phdr)         //计算kernel有多少扇区
{
    uint8_t number = (Phdr->p_filesz + 511) / 512;    //p_filesz表示segment段的大小，但是其可能不是512的整数倍，故向上取整
    return number;
}
```

segment 的大小被记录在 p_filesz 中，将其读取出来后做一个除以 512 的向上取整后可以得到 segment 占用扇区的大小。

③ 将 bootblock 写入 image

```
void write_bootblock(FILE *image, FILE *file, Elf32_Phdr *phdr) //把bootblock写入到image
{
    uint8_t buffer[512];
    int counter;                          //p_offset为segment到文件首地址的偏移量
    fseek(file, phdr->p_offset, SEEK_SET); //  例：fseek(fp,50L,SEEK_SET);    其作用是将位置指针移到离文件首地址50个字节处
                                           //SEEK_SET 文件首地址  SEEK_CUR 当前地址  SEEK_END 文件尾地址
    for( counter = 1; counter*512 <= phdr->p_filesz; counter++)
    {
         memset(buffer, 0, sizeof(buffer));
         fread(buffer, 1, 512, file);
         fwrite(buffer, 1, 512, image);
    }
    if((phdr->p_filesz)%512)                        //如果segment段大小不为整数扇区,就多写一次（包括大小不到一个扇区的情况）
    {
         memset(buffer, 0, sizeof(buffer));
         fread(buffer, 1, 512, file);
         fwrite(buffer, 1, 512, image);
    }

}

//  (1)写操作fwrite()后必须关闭流fclose()。

//   (2)不关闭流的情况下，每次读或写数据后，文件指针都会指向下一个待写或者读数据位置的指针。
```

从之前的描述中我们大致明白了一件事：无论大小为多少，bootblock 与 kernel 都要占据整数个扇区。于是这就涉及到填充（padding），对于大小不满足整数个扇区的部分，我们需要用数据 0 将其填充到整数个扇区。首先我们用 fseek 函数寻找到对应 segment 段起始的位置，随后以 512KB（扇区）为单位进行 fread 和 fwrite 操作。最后，检测是否为 512 的整数倍，如果不是的话就进行拷贝并填充。

④ 将 kernel 写入 image

```
void write_kernel(FILE *image, FILE *knfile, Elf32_Phdr *Phdr, int kernelsz) //把kernel写入到image
{
    uint8_t buffer[512];
    int counter;
    counter = kernelsz;
    fseek(knfile, Phdr->p_offset, SEEK_SET);
    while(counter--)
    {
        memset(buffer, 0, sizeof(buffer));
        fread(buffer, 1, 512, knfile);
        fwrite(buffer, 1, 512, image);
    }
}
```

方法与③相同，只不过由于引入了 kernel 中 segment 扇区数为参变量，循环得到了化简，只需要每次循环后将参量的值减一即可。

⑤ 把 kernel 扇区个数写到 bootblock 中 os_size 的位置

```
void record_kernel_sectors(FILE *image, uint8_t kernelsz)          //把扇区个数写道bootblock的os_size位
{
    uint8_t *buffer;
    buffer = &kernelsz;                                            //指针
    fseek(image, 511, SEEK_SET);                                   //汇编源码
    fwrite(buffer, 1, 1, image);
}
```

首先我们需要明确 bootblock 中哪个部分是表示 os_size 的部分。询问助教老师得到的回答是写在没有数据（打印出来为 0 的部分）就好。总觉得哪里不对，对已经构建好的 createimage 进行反汇编操作后得到的部分代码如下图：

```
08048e0b <write_os_size>:
 8048e0b:       55                      push   %ebp
 8048e0c:       89 e5                   mov    %esp,%ebp
 8048e0e:       83 ec 28                sub    $0x28,%esp
 8048e11:       8b 45 08                mov    0x8(%ebp),%eax
 8048e14:       8d 90 ff 01 00 00       lea    0x1ff(%eax),%edx
 8048e1a:       85 c0                   test   %eax,%eax
 8048e1c:       0f 48 c2                cmovs  %edx,%eax
 8048e1f:       c1 f8 09                sar    $0x9,%eax
 8048e22:       83 e8 01                sub    $0x1,%eax
 8048e25:       66 89 45 f4             mov    %ax,-0xc(%ebp)
 8048e29:       c7 44 24 08 00 00 00    movl   $0x0,0x8(%esp)
 8048e30:       00
 8048e31:       c7 44 24 04 02 00 00    movl   $0x2,0x4(%esp)
 8048e38:       00
 8048e39:       8b 45 0c                mov    0xc(%ebp),%eax
 8048e3c:       89 04 24                mov    %eax,(%esp)
 8048e3f:       e8 fc f6 ff ff          call   8048540 <fseek@plt>
 8048e44:       8d 45 f4                lea    -0xc(%ebp),%eax
 8048e47:       8b 55 0c                mov    0xc(%ebp),%edx
 8048e4a:       89 54 24 0c             mov    %edx,0xc(%esp)
 8048e4e:       c7 44 24 08 01 00 00    movl   $0x1,0x8(%esp)
 8048e55:       00
 8048e56:       c7 44 24 04 02 00 00    movl   $0x2,0x4(%esp)
 8048e5d:       00
```

在地址为 0x08048e0e 处发现，按照传参法则，传入对应 os_size 位置的参数为 511。这意味着偏离源地址 511 比特，也就是 bootblock 最后一比特位置上的参数就是 kernel 的扇区个数。于是用 fseek 指令和 fwrite 指令将其写入。

⑥ 打印 extent-opt 信息

```
void extent_opt(Elf32_Phdr *Phdr_bb, Elf32_Phdr *Phdr_k, int kernelsz) //打印信息
{
    printf("Bootblock message: \n");
    printf("size   %d byte \n", Phdr_bb->p_filesz);
    printf("memory size  0x%x \n",Phdr_bb->p_memsz);
    printf("memory offset  0x%x \n", Phdr_bb->p_offset);
    printf("virtual address  0x%x \n", Phdr_bb->p_vaddr);
    printf("padding up to 0x%x \n\n", 0x200*((Phdr_bb->p_filesz+511)/512));


    printf("Kernel message: \n");
    printf("sector size   %d \n", kernelsz);
    printf("memory size  0x%x \n",Phdr_k->p_memsz);
    printf("memory offset  0x%x \n", Phdr_k->p_offset);
    printf("virtual address  0x%x \n", Phdr_bb->p_vaddr);
    printf("padding up to 0x%x \n\n", 0x200*((Phdr bb->p filesz+511)/512)+0x200*kernelsz);
```

比较普通。记得打印比特使用十进制，打印地址使用十六进制较为方便。另外，打印了填充地址。

⑦ main 函数

```
int main()
{
    int kernelsz;
    FILE *bfile = fopen("bootblock","rb");              //rb只读打开二进制文件
    FILE *kfile = fopen("kernel", "rb");                //打开流
    FILE *image = fopen("image", "wb");                 //wb只写打开二进制文件
    Elf32_Phdr *Phdr_bootblock = read_exec_file(bfile);  //读取bootblock程序头
    Elf32_Phdr *Phdr_kernel = read_exec_file(kfile);     //读取kernel程序头

    kernelsz = (int)count_kernel_sectors(Phdr_kernel);
    write_bootblock(image, bfile, Phdr_bootblock);       //bootblock的segment写入image

    write_kernel(image, kfile, Phdr_kernel, kernelsz);

    record_kernel_sectors(image, kernelsz);

    extent_opt(Phdr_bootblock, Phdr_kernel, kernelsz);
    fclose(bfile);
    fclose(kfile);
    fclose(image);



}
```

使用 fopen 指令分别以只读二进制，只写二进制方式打开了源文件 bootblock（打开到 bfile）、kernel（打开到 kfile）和目标生成文件 image（打开到 image）。分别读去 bfile 和 kfile 的程序头，计算出 kernel 的扇区数，随后执行写函数 write_bootblock 和 write_kernel，最后将 kernel 扇区数写入到 bootblock 区域。最终，打印 extent 项，并关闭文件。

(3) 遇到的问题和解决方法

① fopen 函数和 fclose 函数的使用：

fopen 函数用来打开一个文件（以文件流的形式），其调用的一般形式为：

文件指针名 = fopen（文件名,使用文件方式）

"r" 以只读方式打开文件，该文件必须存在。

"w" 打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件。

"w+" 打开可读写文件,若文件存在则文件长度清为零,即该文件内容会消失。若文件不存在则建立该文件。

"a" 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。(EOF 符保留)

"a+" 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。(原来的 EOF 符不保留)

“rt” 只读打开一个文本文件，只允许读数据

“wt” 只写打开或建立一个文本文件，只允许写数据

“at” 追加打开一个文本文件，并在文件末尾写数据

“rb” 只读打开一个二进制文件，只允许读数据

“wb” 只写打开或建立一个二进制文件，只允许写数据

“ab” 追加打开一个二进制文件，并在文件末尾写数据

“rt+”读写打开一个文本文件，允许读和写

“wt+”读写打开或建立一个文本文件，允许读写

“at+”读写打开一个文本文件，允许读，或在文件末追加数据

“rb+”读写打开一个二进制文件，允许读和写

“wb+”读写打开或建立一个二进制文件，允许读和写

“ab+”读写打开一个二进制文件，允许读，或在文件末追加数据

另外，使用 fopen 打开的文件流，在使用 fclose 函数关闭之前，其文件指针随着读和写不断移动。直到关闭文件后，文件指针才会回到首地址。

fclose() 函数关闭一个打开的文件流。file 参数是一个文件指针。fclose() 函数关闭该指针指向的文件。如果成功则返回 true，否则返回 false。文件指针必须有效，并且是通过 fopen() 或 fsockopen() 成功打开的。

② fwrite 和 fread 函数的使用

fread()函数作用：从一个文件流中读取数据

```
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

buffer:指向数据块的指针

size:每个数据的大小，单位为 Byte(例如：sizeof(int)就是 4)

count:数据个数

stream:文件指针

注意：返回值随着调用格式的不同而不同：

(1) 调用格式：fread(buf,sizeof(buf),1,fp);

读取成功时：当读取的数据量正好是 sizeof(buf)个 Byte 时，返回值为 1(即 count)

否则返回值为 0(读取数据量小于 sizeof(buf))

(2)调用格式：fread(buf,1,sizeof(buf),fp);

读取成功返回值为实际读回的数据个数(单位为 Byte)

size_t fwrite(const void* buffer, size_t size, size_t count, FILE* stream);

buffer:指向数据块的指针

size:每个数据的大小，单位为Byte(例如：sizeof(int)就是4)

count:数据个数

stream:文件指针

注意：返回值随着调用格式的不同而不同：

(1) 调用格式：fwrite(buf,sizeof(buf),1,fp);

成功写入返回值为1(即count)

(2)调用格式：fwrite(buf,1,sizeof(buf),fp);

成功写入则返回实际写入的数据个数(单位为Byte)

③ memcpy函数的使用

void *memcpy(void *dest, void *source, unsigned n);

dest：指向用于存储复制内容的目标数组（类型强制转换为void）

source：指向要复制的数据源（类型强制转换为void）

n:　要被复制的字节数

返回一个指向目标存储区dest的指针。

从源source所指的内存地址的起始位置开始拷贝n个字节到目标dest所指的内存地址的起始位置中。

④ uint8_t

uint8_t不是一种新变量类型，而是由以下定义：

typedef unsigned char uint8_t;

这说明其为占用一个字节的无符号char型变量。

所以在打印该值的时候可能会有一些有趣的事情发生。

附：

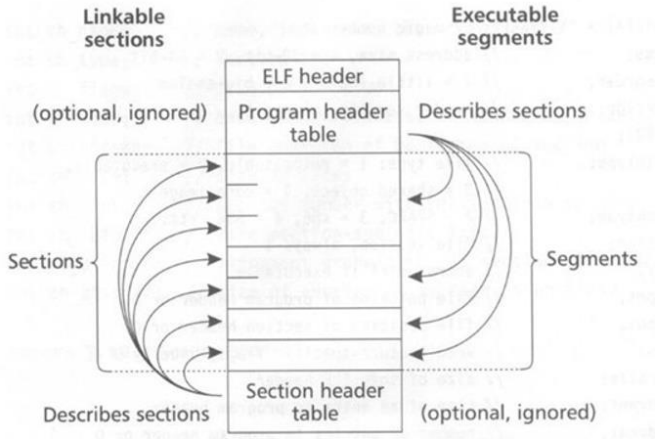
```
typedef unsigned int uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;

typedef signed char int8_t;
typedef int int16_t;
typedef long int32_t;
typedef long long int64_t;

typedef int16_t intptr_t;
typedef uint16_t uintptr_t;
```

⑤ 进一步的理解

ELF 二进制可执行文件结构：



ELF header 有 52 个字节，描述整个 ELF 文件：

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];   /* Magic number and other info */
  Elf32_Half    e_type;               /* Object file type */
  Elf32_Half    e_machine;            /* Architecture */
  Elf32_Word    e_version;            /* Object file version */
  Elf32_Addr    e_entry;              /* Entry point virtual address */
  Elf32_Off     e_phoff;              /* Program header table file offset */
  Elf32_Off     e_shoff;              /* Section header table file offset */
  Elf32_Word    e_flags;              /* Processor-specific flags */
  Elf32_Half    e_ehsize;             /* ELF header size in bytes */
  Elf32_Half    e_phentsize;          /* Program header table entry size */
  Elf32_Half    e_phnum;              /* Program header table entry count */
  Elf32_Half    e_shentsize;          /* Section header table entry size */
  Elf32_Half    e_shnum;              /* Section header table entry count */
  Elf32_Half    e_shstrndx;           /* Section header string table index */
} Elf32_Ehdr;
```

描述程序段的 Program header：

```
typedef struct
{
  Elf32_Word    p_type;         /* Segment type */
  Elf32_Off     p_offset;       /* Segment file offset */
  Elf32_Addr    p_vaddr;        /* Segment virtual address */
  Elf32_Addr    p_paddr;        /* Segment physical address */
  Elf32_Word    p_filesz;       /* Segment size in file */
  Elf32_Word    p_memsz;        /* Segment size in memory */
  Elf32_Word    p_flags;        /* Segment flags */
  Elf32_Word    p_align;        /* Segment alignment */
} Elf32_Phdr;
```

描述 segment 段中每个节（section）的 section header：

```
typedef struct
{
  elf32_word    sh_name;        /* Section name (string tbl index) */
  elf32_word    sh_type;        /* Section type */
  elf32_word    sh_flags;       /* Section flags */
  elf32_addr    sh_addr;        /* Section virtual addr at execution */
  elf32_off     sh_offset;      /* Section file offset */
  elf32_word    sh_size;        /* Section size in bytes */
  elf32_word    sh_link;        /* Link to another section */
  elf32_word    sh_info;        /* Additional section information */
  elf32_word    sh_addralign;   /* Section alignment */
  elf32_word    sh_entsize;     /* Entry size if section holds table */
} elf32_shdr;
```

⑥ p_filesz 与 p_memsz 的关系

"The `p_filesz` *field corresponds to the segment's size in bytes in the file, whereas the* `p_memsz` *is the segment's in-memory size. The reason why* `p_memsz` *is greater than (or equal to)* `p_filesz` *is that a loadable segment may contain a* `.bss` *section, which contains uninitialized data. It would be wasteful to store this data on disk, and therefore it only occupies space once the ELF file is loaded into memory. This fact is indicated by the* `SHT_NOBITS` *type of the* `.bss` *section."*

意思就是说 p_filesz 描述的是不含未初始化的变量的文件大小，在写入内存的时候这些变量的值才会被从 bss 段中加载到数据段中。

(4) 实现代码段

① bootblock.c main 函数段

与任务一相同

② kernel.c

```
void (*printstr)(char *str);
char  value[] = "2019";
void __attribute__((section(".entry_function"))) _start(void)
{
        // Call PMON BIOS printstr to print message "Hello OS!"

        char message[] = "Hello OS!\n";
        char v[] = "Version:";
        printstr = (void *)0x80011100;
        (*printstr)(message);
        (*printstr)(v);
        (*printstr)(value);

        return;
}
```

③ createimage.c

详见（3）

（5） 运行结果

MAKE ALL：

```
stu@stu-VirtualBox:~/project1/create$ make all
mipsel-linux-gcc -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=0xffffffffa0800000
 -N -o bootblock bootblock.s -nostdlib -e main -Wl,-m -Wl,elf32ltsmip -T ld.script
mipsel-linux-gcc -G 0 -O2  -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=0xffffffffa0800200
 -N -o kernel kernel.c -nostdlib -Wl,-m -Wl,elf32ltsmip -T ld.script
./createimage --extended bootblock kernel
Bootblock message:
size   144 byte
memory size  0x90
memory offset  0x60
virtual address  0xa0800000
padding up to 0x200

Kernel message:
sector size   1
memory size  0x114
memory offset  0x60
virtual address  0xa0800000
padding up to 0x400
```

LOADBOOT：

```
03 00 00 00 88 00 80 a0 e8 00 00 00 08 00 00 00
00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
11 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00
f0 00 00 00 35 00 00 00 00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00
00 00 00 00 00 00 00 00 68 02 00 00 c0 01 00 00
07 00 00 00 0c 00 00 00 04 00 00 00 10 00 00 00
09 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00
28 04 00 00 c1 00 00 00 00 00 00 00 00 00 00 01
It's a bootloader...
Hello OS!
Version:2019
```