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。

1. 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

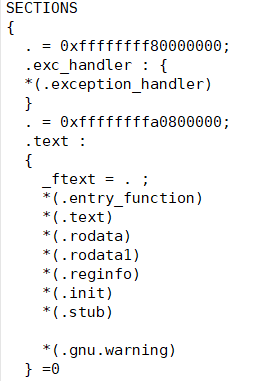
通过la指令将要打印的字符串地址送至传参寄存器$a0，然后通过jal指令跳转值printstr函数的地址0x80011100执行打印字符串

* 1. 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。

* 1. 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。

1. 遇到的问题和解决方法
2. 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
3. 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 = "

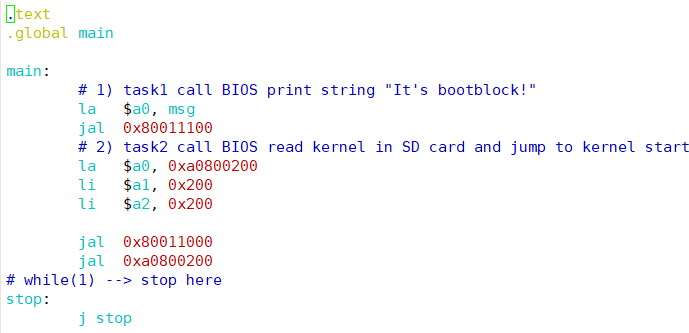
……

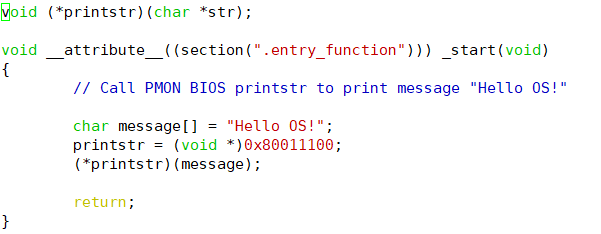
la $a1, output

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。

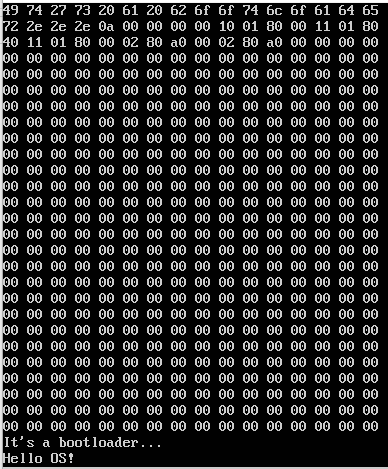
kernel : .word 0xa0800200

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。

1. C语言对函数的调用，可由函数指针语句实现。
2. 在全局区域声明 void ( \*函数名)（参数类型 参数名）
3. 在main中声明 函数名 = （void）函数地址
4. 调用 （\*函数名）（参数）
5. 实现代码段
   1. bootblock.c main函数段
   2. kernel.c



1. 运行结果



**Bootloader bonus**

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。

1. 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。

1. 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

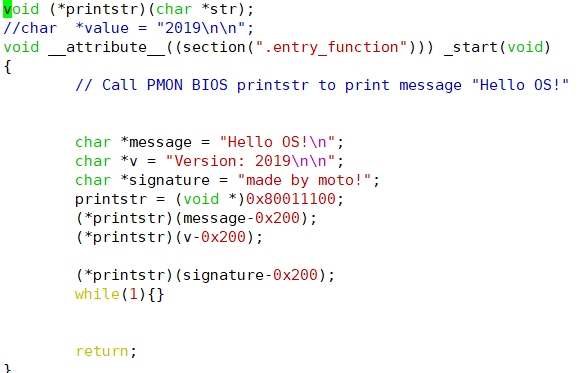
${CC} -G 0 -O2 -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=0xffffffffa0800000 -N -o bootblock bootblock.s -nostdlib -e main -Wl,-m -Wl,elf32ltsmip -T ld.script

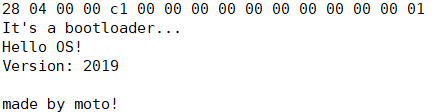
kernel: kernel.c

${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。

* 1. 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。

1. 实现代码段
   1. bootblock.c main函数段
   2. kernel.c
2. 运行结果



**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。

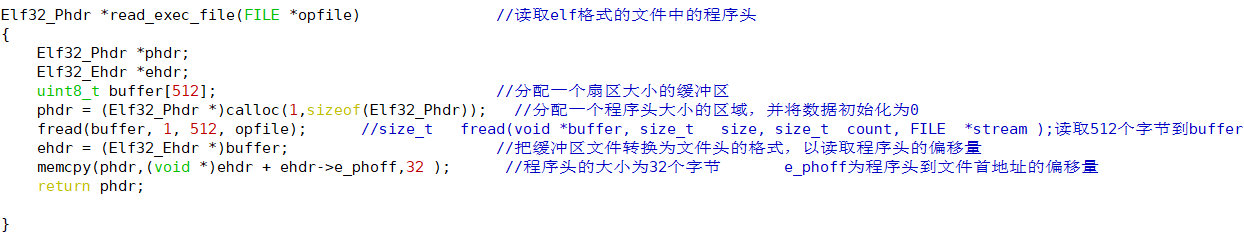
image文件由两部分组成：

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。

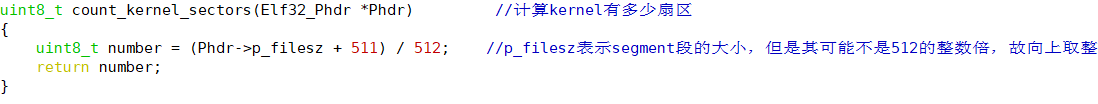
1. Createimage的执行过程

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：

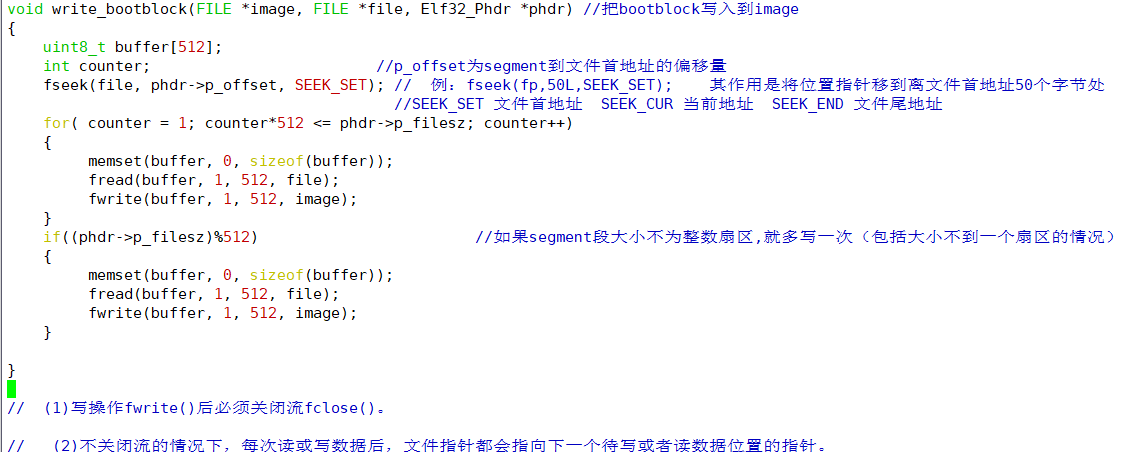
* 1. 读取ELF文件中的程序头（Program segment header）

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。

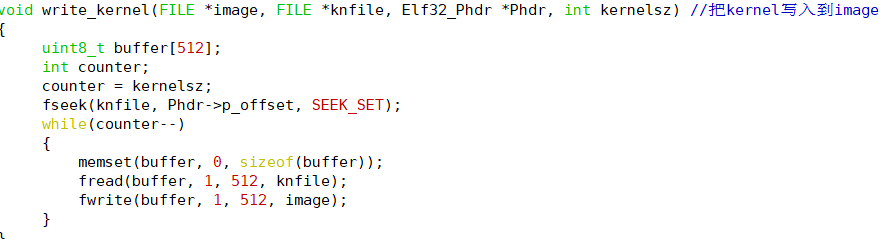
* 1. 计算kernel的segment段有多少扇区



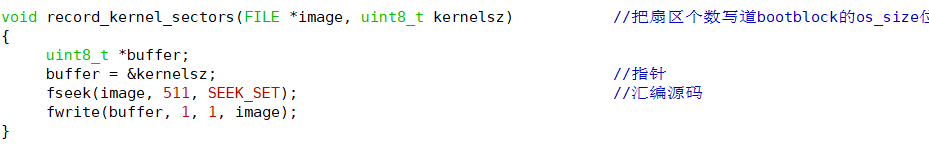
segment的大小被记录在p\_filesz中，将其读取出来后做一个除以512的向上取整后可以得到segment占用扇区的大小。

* 1. 将bootblock写入image

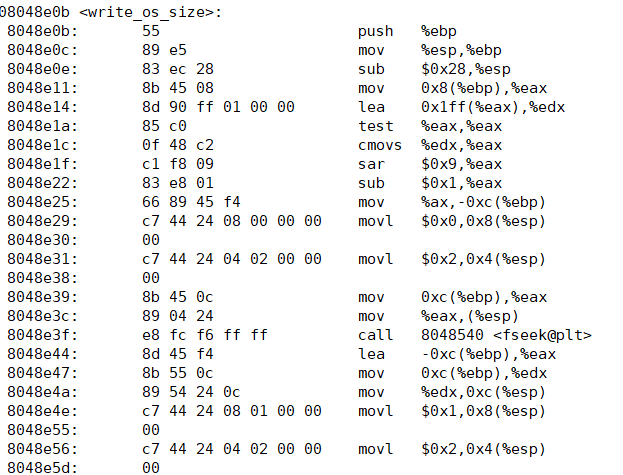
从之前的描述中我们大致明白了一件事：无论大小为多少，bootblock与kernel都要占据整数个扇区。于是这就涉及到填充（padding），对于大小不满足整数个扇区的部分，我们需要用数据0将其填充到整数个扇区。首先我们用fseek函数寻找到对应segment段起始的位置，随后以512KB（扇区）为单位进行fread和fwrite操作。最后，检测是否为512的整数倍，如果不是的话就进行拷贝并填充。

* 1. 将kernel写入image

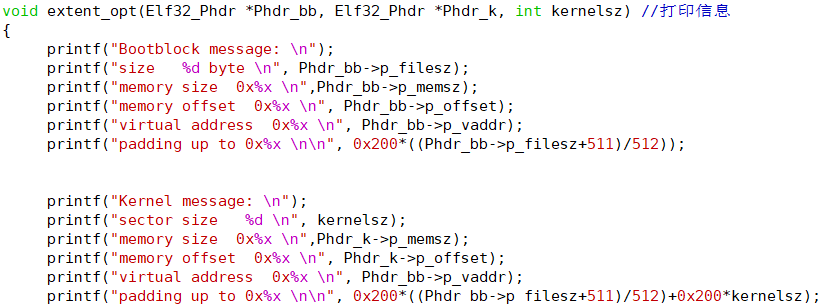
方法与③相同，只不过由于引入了kernel中segment扇区数为参变量，循环得到了化简，只需要每次循环后将参量的值减一即可。

* 1. 把kernel扇区个数写到bootblock 中os\_size的位置

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：



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。

* 1. 打印extent-opt信息

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。

* 1. main函数

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。

1. 遇到的问题和解决方法
2. 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](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=8417547&ss_c=ssc.citiao.link)，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。

1. 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)

1. 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。

1. 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；

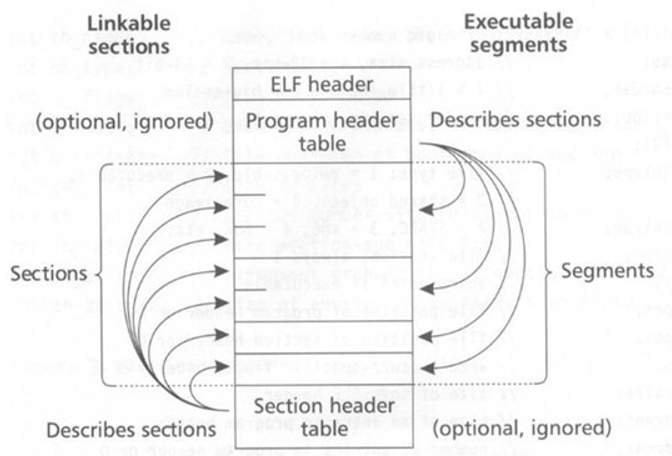
typedef long int32\_t；

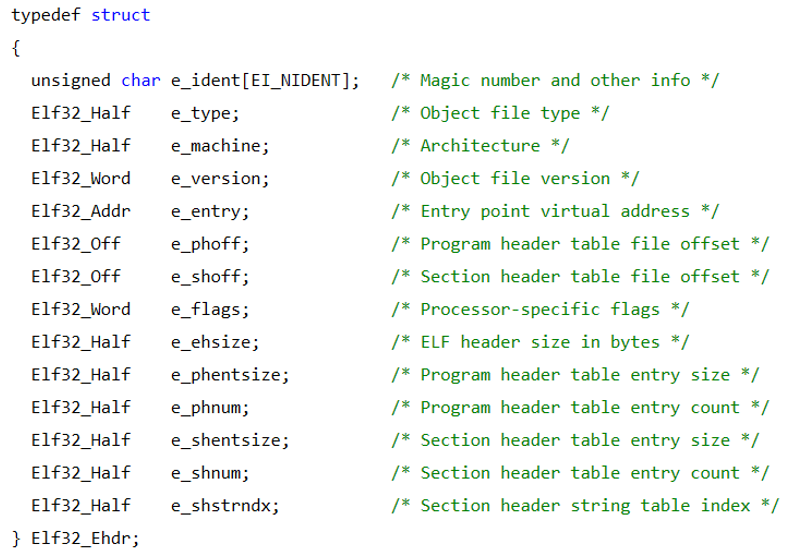
typedef long long int64\_t；

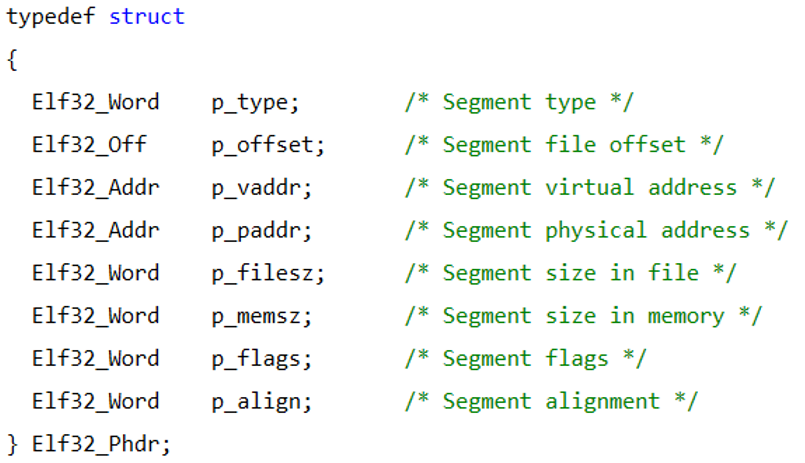
typedef int16\_t intptr\_t;

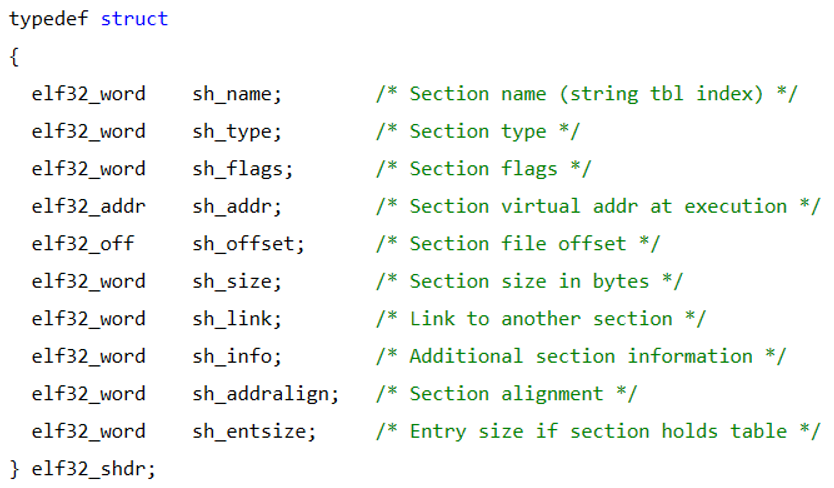
typedef uint16\_t uintptr\_t;

1. 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：

描述程序段的Program header：

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

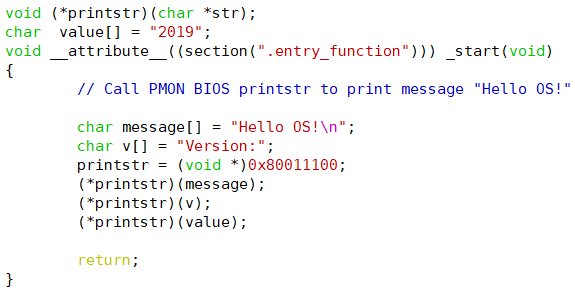
1. p\_filesz 与p\_memsz的关系

“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 .bsssection, 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。

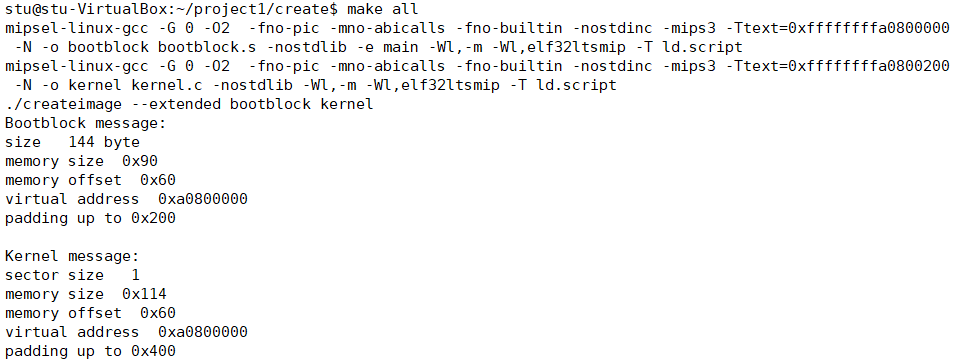
1. 实现代码段
   1. bootblock.c main函数段

与任务一相同

* 1. kernel.c
  2. createimage.c

详见（3）

1. 运行结果

make all：

loadboot：

