[GRUB启动分析之stage1.5](http://blog.chinaunix.net/uid-24774106-id-3500759.html) 2013-02-26 22:45:53

分类： LINUX

**前言**

上一篇博文介绍了GRUB源码的stage1.S会汇编成一段446字节的sourcecode，stage1,grub会将这个stage1放入MBR中。我们通过分析，知道这段代码的唯一作用就是将第二个扇区（0柱面 0 磁道 2扇区）处的512字节加载到内存中去。

一个问题就来了这个512个字节是从何而来，这512个字节又意欲何为？江湖上风传已久的stage1.5是什么东东，stage2又是干什么的？ 本文将要解释这些内容。

**start.S源码分析**

加载到内存的第二个扇区的内容是由GRUB源码stage/start.S汇编而成，这个汇编文件汇编出来的二进制文件大小也是512B，一个扇区的大小。stage1阶段结束后，这段代码就被加载进了内存。那么这段代码是干啥的呢？

我先透个底，整个start.S代码的作用是从LBA 扇区号2（0柱面 0磁道 3扇区） 开始拷贝若干个扇区到内存。起始扇区号，扇区个数 内存目的地址都在start.S中定义了。下面我们看下代码：

lastlist:

/\*

\* This area is an empty space between the main body of code below which

\* grows up (fixed after compilation, but between releases it may change

\* in size easily), and the lists of sectors to read, which grows down

\* from a fixed top location.

\*/

.word 0

.word 0

. = \_start + 0x200 - BOOTSEC\_LISTSIZE

/\* fill the first data listing with the default \*/

blocklist\_default\_start:

.long 2 /\* this is the sector start parameter, in logical

sectors from the start of the disk, sector 0 \*/

blocklist\_default\_len:

/\* this is the number of sectors to read \*/

#ifdef STAGE1\_5

.word 0 /\* the command "install" will fill this up \*/

#else

.word (STAGE2\_SIZE + 511) >> 9

#endif

blocklist\_default\_seg:

#ifdef STAGE1\_5

.word 0x220

#else

.word 0x820 /\* this is the segment of the starting address

to load the data into \*/

#endif

firstlist: /\* this label has to be after the list data!!! \*/

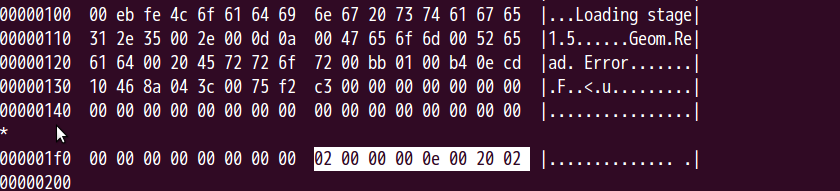
firstlast是扇区后第一个字节，start.S对应的二进制文件对应于第二个扇区，也即（512-1024）字节。那么firstlast就是第二个扇区后的第一位置，即磁盘的1024位置处，第二扇区的512字节处。

blocklist\_default\_seg占2个字节，blocklist\_default\_len占两个字节，blocklist\_default\_start占四个字节，那么我们可以算出（firstlist-8）的地址就是blocklist\_default\_start的地址。这个位置记录的是起始扇区。我们看到起始扇区是2,LBA模式扇区number 2对应的是（0柱面 0磁道 3扇区）。至于blocklist\_default\_len，注释用有提到，install的时候，填写这个数字。

下面我们看下我们磁盘的start.S对应的二进制文件，我们知道，start.S对应磁盘512～1024部分。

dd if=/dev/sda of=mbr\_512\_1024 bs=512 skip=1 count=1

现在我们看看这个start.S汇编出来的二进制文件是：



按照我们当前的分析。firstlist对应上图中的0x200位置，那么blocklist\_default\_seg就是最后2个字节0220h。从此处可以看出，对于我们的start.S中#ifdef STAGE1\_5这个宏是打开的。另外我们可以从上图中的loading stage1.5可以看出。

#ifdef STAGE1\_5

notification\_string: .string "Loading stage1.5"

#else

notification\_string: .string "Loading stage2"

#endif

从我们打印出来的2进制文件可以看出，起始LBA扇区号blocklist\_default\_start是2,blocklist\_default\_len是0x0e。

现在我们闲言少叙，继续分析代码：

movw $ABS(firstlist - BOOTSEC\_LISTSIZE), %di

/\* save the sector number of the second sector in %ebp \*/

movl (%di), %ebp

/\* this is the loop for reading the secondary boot-loader in \*/

bootloop:

/\* check the number of sectors to read \*/

cmpw $0, 4(%di)

/\* if zero, go to the start function \*/

je bootit

就如刚才分析的，将blocklist\_default\_start对应的地址存到di寄存器中，同时将blocklist\_default\_start位置存储的值2存储到ebp寄存器。 blocklist\_default\_start+4 = blocklist\_default\_len的地址。那么cmpw这个指令比较的是需要copy的扇区个数，对于我们的例子是比较0x0e和0,既然不相等，那么不跳转，继续执行：

setup\_sectors:

/\* check if we use LBA or CHS \*/

cmpb $0, -1(%si)

/\* jump to chs\_mode if zero \*/

je chs\_mode

lba\_mode:

/\* load logical sector start \*/

movl (%di), %ebx

/\* the maximum is limited to 0x7f because of Phoenix EDD \*/

xorl %eax, %eax

movb $0x7f, %al

/\* how many do we really want to read? \*/

cmpw %ax, 4(%di) /\* compare against total number of sectors \*/

/\* which is greater? \*/

jg 1f

/\* if less than, set to total \*/

movw 4(%di), %ax

这段代码的含义是，先判断走那个分支，对于我们是LBA mode，然后比较需要copy的扇区个数是否高于0x7f，如果高于7fh个，要分批次copy。对于我们而言，不需要因为我们4（%di）处存放的值是0xe。我们将0xe存储到ax寄存器

1:  
/ subtract from total / subw %ax, 4(%di)

/\* add into logical sector start \*/

addl %eax, (%di)

/\* set up disk address packet \*/

/\* the size and the reserved byte \*/

movw $0x0010, (%si)

/\* the number of sectors \*/

movw %ax, 2(%si)

/\* the absolute address (low 32 bits) \*/

movl %ebx, 8(%si)

/\* the segment of buffer address \*/

movw $BUFFERSEG, 6(%si)

/\* save %ax from destruction! \*/

pushw %ax

/\* zero %eax \*/

xorl %eax, %eax

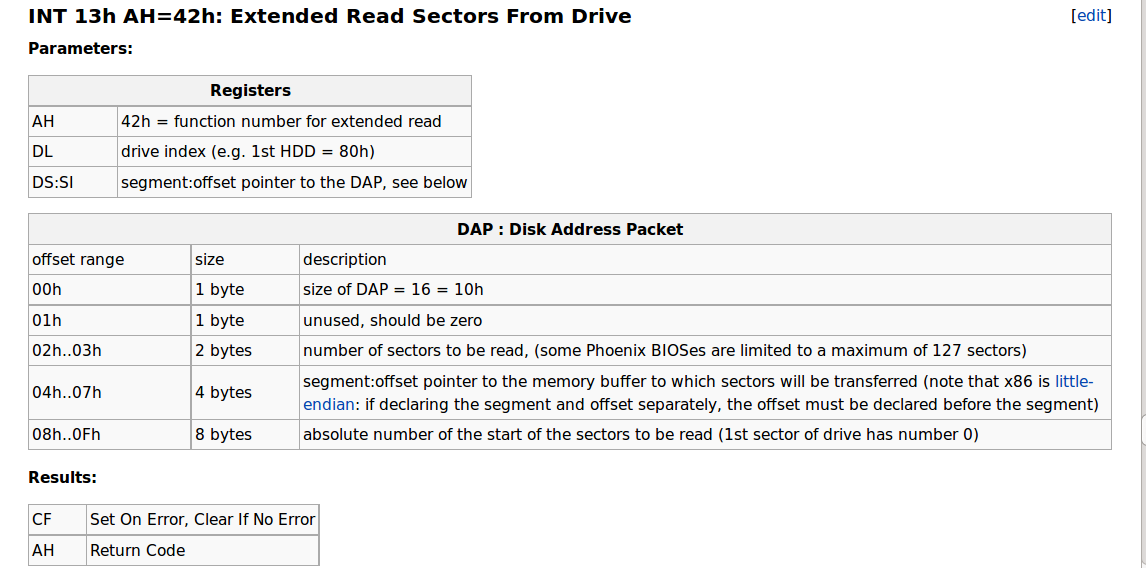
/\* the offset of buffer address \*/

movw %ax, 4(%si)

/\* the absolute address (high 32 bits) \*/

movl %eax, 12(%si)

这部分工作是做INT 13H（FUNCTION 0x42）的准备工作。我们在GRUB分析之stage1中做过一次分析了。我们再次分析下：



si[0] -----10h

si[1] -----00h

s[2 3] -----扇区个数0xe

s·[6 7]-----拷贝到的内存位置 $BUFFERSEG=0x7000

s[08h..0Fh]----LBA起始扇区号 对于我们是0x2

继续分析，就是到了执行INT 13H

movb $0x42, %ah

int $0x13

jc read\_error

movw $BUFFERSEG, %bx

jmp copy\_buffer

这不细说了，就是拷贝扇区内容到$BUFFERSEG内存处。

最后到了copy\_buffer,作用和上一篇一样。movsb指令将ds:si（0x7000：0x0000）处连续的 %bx字节内容传输到es:di指定的内存地址（0×2200:0×0000）。其中，rep指令的含义就是重复执行后一句指令，每执行一次。cx减1，直至cx为0。好了，从头慢慢来，首先将6（%di）上的0x220赋给%es，就是目的段地址了；然后取得ax的值，ax的值是本次读出的扇区数，左移5位并且把值赋给6（%si），这就是下一次循环执行这个程序时新的目的段地址了（对于我们只有0xege扇区自然是不需要了）。然后保存ds的值，接着再将ax左移4位，相当于一共左移了9位，也就是将ax的值乘以512，然后将这个值赋给cx寄存器，正是要传送的字节数。

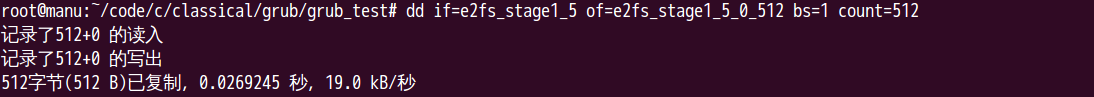
由于我们的根文件系统是ext3文件系统，我们发现在/boot/grub/下除了stage1,还有e2fs\_stage1\_5这个文件，这个文件的作用是识别ext3文件系统的。我们知道GRUB开始没有OS，也没有文件系统的概念。那么GRUB是从何时开始有文件系统的功能的呢。这就是stage1.5干的事情，stage1.5过后，GRUB就能识别文件系统了，就能在磁盘上识别加载文件了。怎么做到的？start.S加载的磁盘上的那些扇区的内容，就是文件系统的代码。把这0xe也就是14个扇区的内容加载到内存后，就具备了操作启动设备上面文件的功能了。

但是文件系统千千万，我们不可能把所有文件系统的功能文件放在磁盘的扇区里面，那怎么办呢？grub 执行setup的时候，能够识别启动设备的文件系统，比如我们，是ext3文件系统，所以只需要将ext3部分的e2fs\_stage1\_5放入扇区。

如何证明？

事实上，e2fs\_stage1\_5的前512个字节的内容是start.S对应的二进制文件，后面的部分是操作文件系统部分。我们证明下：

准备活动：



root@manu:~/code/c/classical/grub/grub\_test# dd if=mbr\_1024\_ of=mbr\_1024\_8096 bs=1 count=7072

记录了7072+0 的读入

记录了7072+0 的写出

7072字节(7.1 kB)已复制，0.0478542 秒，148 kB/秒

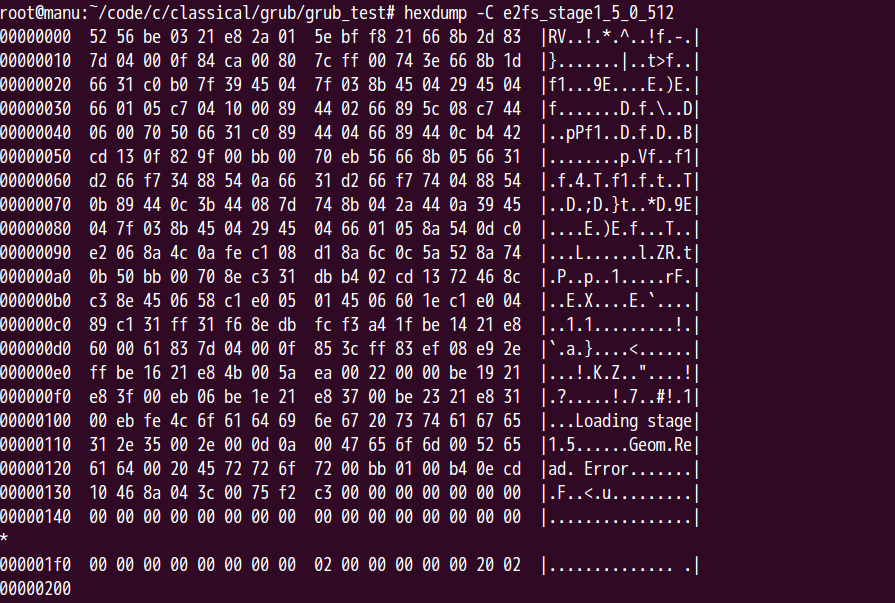
root@manu:~/code/c/classical/grub/grub\_test# dd if=e2fs\_stage1\_5 of=e2fs\_stage1\_5\_512\_7584 bs=1 skip=512 count=7072

记录了7072+0 的读入

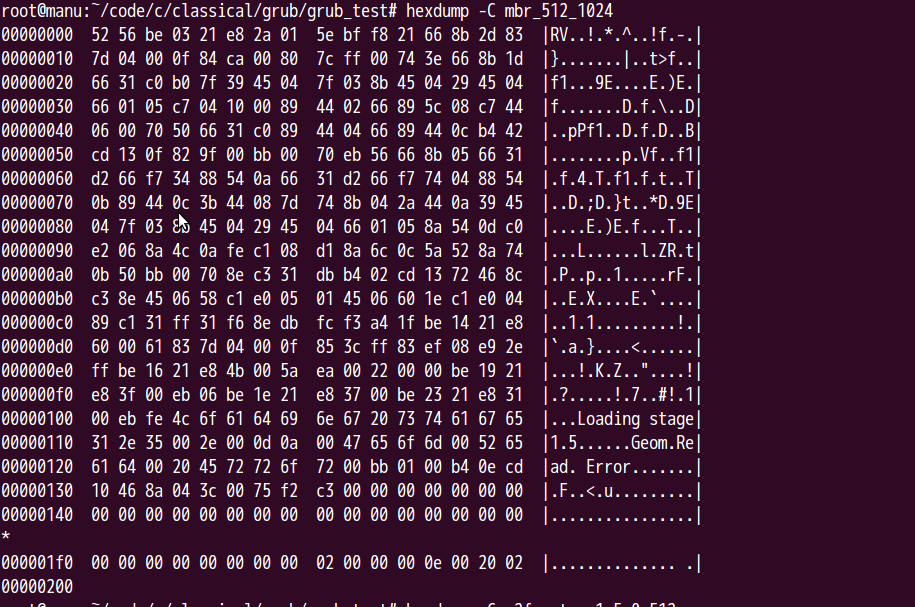
记录了7072+0 的写出

7072字节(7.1 kB)已复制，0.0239908 秒，295 kB/秒

看下e2fs\_stage1\_5前512字节：

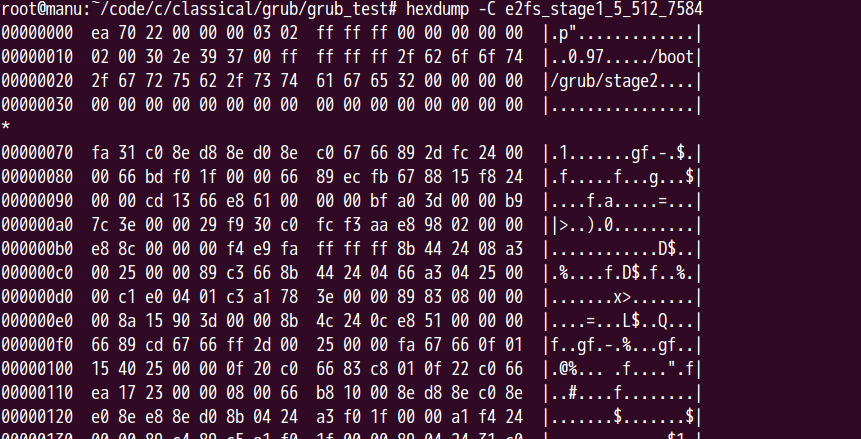


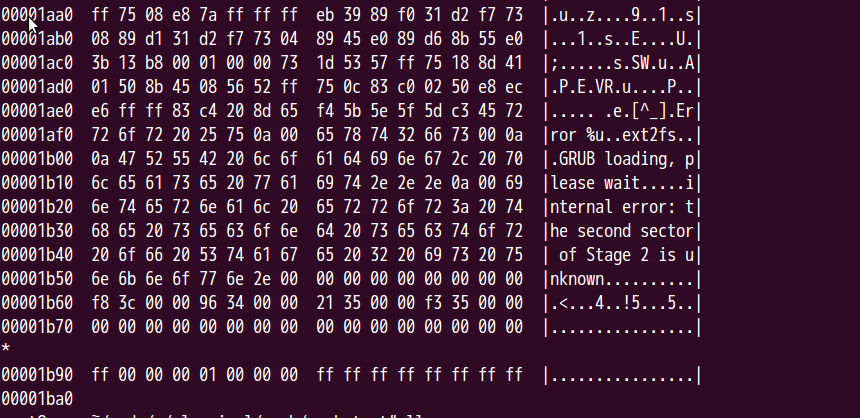
在看下磁盘的第二个扇区，也就是start.S对应的二进制文件：



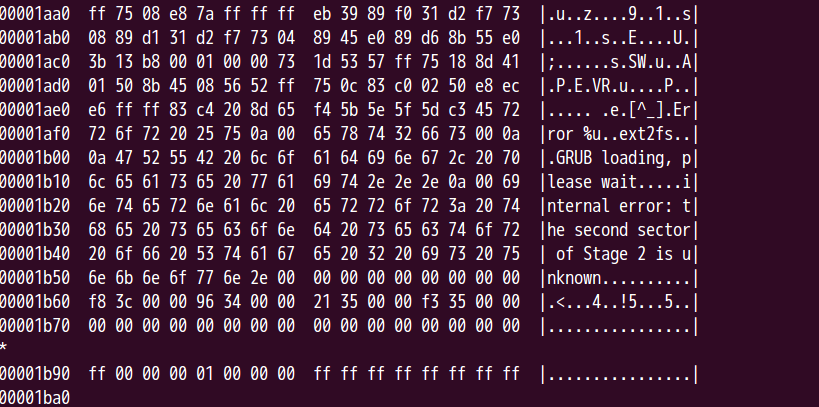
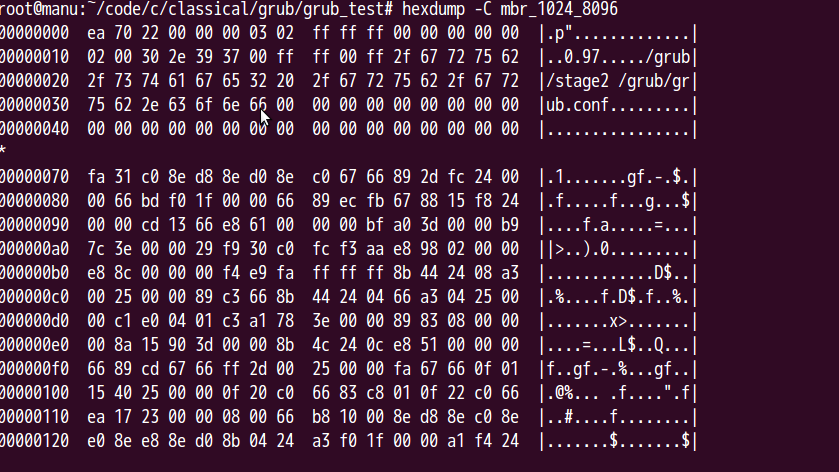
所有的部分都是相同的，只有最后一行的0x1fc位置不同，那一个位置我们已经讨论过了，blocklist\_default\_len，就是需要拷贝扇区的个数。e2fs\_stage1\_5里面是0,磁盘第二个扇区是0xe，就像代码中注释中提到的/ the command "install" will fill this up /，这个值是install的时候填充的。

在看下e2fs\_stage1\_5文件的512字节到最后的内容：





看下磁盘从第三个扇区开始，共7072字节的内容：



比较一下可以得知,除了头部的/boot/grub那些路径不同,其余部分都是相同的。

**总结**

通过这两篇文章我们知道了一下我们获取的信息：

1. MBR code 部分和/boot/grub/stage1部分一样，这部分二进制文件是有grub源代码中的/stage1/stage1.S汇编出的。所谓的stage1,作用只有一个，就是将磁盘第二个扇区的内容加载的到内存
2. 第二个扇区的内容和/boot/grub/e2fs\_stage1\_5文件的前512字节一样，这部分内容是有grub 源码/stage2/start.S汇编出的，而这个start.S的作用就是加载磁盘的第三个扇区到第N个扇区到内存，N取几，取决与文件系统的支撑代码的大小。
3. 文件系统支撑代码到内存之后，我们在也不需要直接调用INT 13加载扇区内容，我们有了文件系统，我们可以直接操作文件了。那么/boot/grub/stage2这样的比较大的文件可以直接操作了。