linux操作系统的启动过程

by zenhumay

2011-11-26——2011-12-11

目录

1. 概述

启动（bootstrap）在操作系统中表示系统在加电开始之后，将操作系统内核镜像和根文件系统（如果存在的话）加载到内存中，同时初始化系统运行环境的过程。

系统的启动从加电那一刻开始，当按下开机按钮后，一个特殊的硬件电路在CPU 引脚上产生一个 RESET 信号，CPU 接受到信号后，将 CS 和 EIP 设置成固定的值，并执行在物理地址 0xffffffff0 处得代码。硬件将这个物理地址映射到BIOS。BIOS 启动过程主要执行以下 4 个操作：

- 测试计算机硬件，用来检测计算机有哪些设备以及这些设备是否正常工作。这个阶段通常称为 POST（上电自检）。
- 初始化硬件设备，保证所有的硬件设备操作不会引起 IRQ 线和 I/O 端口的冲突。
- 搜索一个操作系统来启动。根据 BIOS 的设置，该过程可能要试图访问系统中的软盘、硬盘和 CD-ROM 的第一个扇区（引导扇区）。
- 找到一个有效的设备后，就把第一个扇区的内容拷贝到 RAM 中从物理地址 0x00007c00 开始的地方，然后跳转到这个地方，开始执行加载进来的代码。

本文所要讨论的问题就是 BIOS 将引导扇区的内容拷贝到 0x00007c00 之后的事情。

linux 从最初的发展到如今，引导启动的方式也有基本经历了原始启动、LILO 引导、GRUB 引导三个阶段。

原始启动阶段：操作系统按引导扇区、SETUP 模块、内核镜像的顺序将它们都保存在从引导扇区（0 柱面 0 磁道 1 扇区）开始的一片连续的硬盘空间上，引导扇区加载 SETUP 模块和内核镜像时从固定的硬盘地址读取它们既可。

LILO 启动阶段：LILO 经历了多个版本的进化发展到如今，已经可以支持多系统启动，系统启动所需的内核镜像、根文件系统(initrd)、映射文件可以存放在硬盘的任何地方。LILO 启动除了第一阶段的引导代码（first stage）必须写入 MBR 或者引导扇区外，其余的部分可以存放在硬盘的任意位置。

GRUB 启动阶段：虽然 LILO 启动阶段已经可以满足大部分人的需求，但是由于每次重新编译内核或者移动内核的位置、以及修改配置文件后，都必须运行 LILO 的安装程序，重新生成映射文件并重写引导扇区，才可以保证下次启动中

会正确的引导系统内核，造成这些的大部分原因是因为 LILO 不支持文件系统。GRUB 可以算是在 LILO 上的一个改进，最主要的改进是 LILO 支持文件系统的功能。

2. 原始启动

下面的内容都是基于内核源代码 0.11 版展开的。

linux 0.11 内核的启动文件包括三个：bootsect.s，head.s，setup.s 三个文件：

bootsect.s：磁盘引导块程序，驻留在磁盘的第一个扇区中（0 柱面、0 磁头、1 扇区）。其主要功能是将 setup 模块从磁盘加载到内存 0x90200 处，将 system 模块加载到内存 0x10000 出开始的地方，然后跳转到 0x90200 出开始执行。

setup 模块：该模块的主要功能是利用 ROM BIOS 中断读取机器系统数据，将这些数据保存到 0x90000 开始的位置。调整内核程序到合适的内存。加载中断描述表寄存器(idtr)和全局描述符表寄存器(gdtr)，开启 A20 地址线，重新设置中断控制器芯片。最后设置 CPU 的控制寄存器 CR0，进入 32 位保护模式运行，并跳转到位于 system 模块最前面部分的 head.s 程序继续运行。

head.s 模块：该程序编译后，会被连接成 system 模块最前面的开始部分。

该小结主要讨论 bootsect.s 模块，因为以后的 LILO 和 GRUB 实现的主要是该模块的功能。

2.1 BIOS系统调用

INT 13H
功能02H
功能描述：读扇区
入口参数：AH＝02H
AL＝扇区数
CH＝柱面
CL＝扇区
DH＝磁头
DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘
ES:BX＝缓冲区的地址
出口参数：CF＝0——操作成功，AH＝00H，AL＝传输的扇区数，否则，AH＝状态代码，参见功能号 01H 中的说明

功能08H
功能描述：读取驱动器参数
入口参数：AH＝08H
DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘
出口参数：CF＝1——操作失败，AH＝状态代码，参见功能号01H中的说明，否则， BL＝01H — 360K
＝02H — 1.2M
＝03H — 720K
＝04H — 1.44M
CH＝柱面数的低8位
CL的位7-6＝柱面数的高2位
CL的位5-0＝扇区数
DH＝磁头数
DL＝驱动器数
ES:DI＝磁盘驱动器参数表地址

2.2 bootsect.s源码分析

2.2.1 常量定义

```
34 SETUPLEN = 4                ! nr of setup-sectors
35 BOOTSEG  = 0x07c0           ! original address of boot-sector
36 INITSEG  = 0x9000           ! we move boot here - out of the way
37 SETUPSEG = 0x9020           ! setup starts here
38 SYSSEG   = 0x1000           ! system loaded at 0x10000 (65536).
39 ENDSEG   = SYSSEG + SYSSIZE ! where to stop loading
```

2.2.2 将bootsect.s从内存 0x07c0:0000 搬移到 0x9000:0000

当 BIOS 将 bootsect.s 加载到 0x7c00:0000 出后，跳转到该地址执行的第一条语句就是 47 行的代码。

```
45 entry start
46 start:
47     mov ax,#BOOTSEG
```

```
48      mov ds,ax
49      mov ax,#INITSEG
50      mov es,ax
51      mov cx,#256
52      sub si,si
53      sub di,di
54      rep
55      movw
```

上述代码的主要功能就是将内存 0x07c0:0000-0x07e0:0000 带代码搬移到 0x9000:0000-0x9020:0000 的地方。

2.2.3 加载SETUP代码

```
67 load_setup:
68      mov dx,#0x0000          ! drive 0, head 0
69      mov cx,#0x0002          ! sector 2, track 0
70      mov bx,#0x0200          ! address = 512, in INITSEG
71      mov ax,#0x0200+SETUPLEN ! service 2, nr of sectors
72      int 0x13                ! read it
```

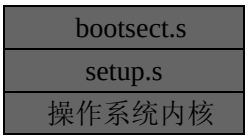
将硬盘上从 0 柱面、0 磁道、2 扇区开始的 SETULEN（4 个）扇区加载到内存 0x9020:0000 开始的内存块。

2.2.4 加载SYSTEM代码

```
107     mov ax,#SYSSEG
108     mov es,ax           ! segment of 0x010000
109     call    read_it
```

将操作系统镜像从 0 柱面、0 磁道、6 扇区开始的 384 个扇区加载到以 0x1000:0000 开始的内存块。

2.3 操作系统在磁盘和内存中的布局图



3. LILO

LILO（LInux LOader）代表 Linux 的加载程序，其主要功能是将是将操作系统的引导模块加载到 0x9000:0000，SETUP 模块加载到 0x9020:0000，根据内核映像的大小，将 zImage（make zImage 生成的内核）加载到 0x1000:0000 出，将 bzImage(make bzImage 生成的内核加载到)0x00100000 出，然后跳转到 0x9020:0000 出，将控制权交给 SETUP 模块。

从上面的分析可以看出，LILO 实现的功能和原始启动中 bootsect.s 功能一致。不过从原始启动的 bootsect.s 功能中可以看出，setup.s 和内核镜像都必须放在磁盘固定的位置才可以引导。

LILO 解决了上述这个问题，它提供充分的灵活性：

操作系统镜像可以存放在磁盘的任何地方

可以加载根文件系统

支持多操作系统的启动。

本节以下的内容基于 LILO 22.8 源码展开

3.1 扩展BIOS系统调用

3.1.1 扩展BIOS的目的

设计扩展 INT 13H 接口的目的是为了扩展 BIOS 的功能，使其支持多于 1024 柱面的硬盘。

3.1.2 磁盘地址数据包Disk Address Packet(DAP)

DAP 是基于绝对扇区地址的，因此利用 DAP，INT 13H 可以轻松地逾越 1024 柱面的限制，其根本不需要 CHS 的概念。

DAP 的结构如下：

```
struct DiskAddressPacket
{
BYTE PacketSize; // 数据包尺寸(16 字节)
BYTE Reserved; // ==0
WORD BlockCount; // 要传输的数据块个数(以扇区为单位)
DWORD BufferAddr; // 传输缓冲地址(segment:offset)
QWORD BlockNum; // 磁盘起始绝对块地址
};
```

PacketSize：保存了 DAP 结构的尺寸，以便将来对其进行扩充. 在目前使用的扩展 Int13H 版本中 PacketSize 恒等于 16. 如果它小于 16, 扩展 Int13H 将返回错误码(AH=01,CF=1).

BlockCount：对于输入来说是需要传输的数据块总数, 对于输出来说是实际传输的数据块个数. BlockCount = 0 表示不传输任何数据块.

BufferAddr：是传输数据缓冲区的 32 位地址(段地址:偏移量). 数据缓冲区必须位于常规内存以内(1M).

BlockNum：表示的是从磁盘开始算起的绝对块地址(以扇区为单位),与分区无关. 第一个块地址为 0. 一般来说, BlockNum 与 CHS 地址的关系是:

BlockNum = cylinder * NumberOfHeads +head * SectorsPerTrack + sector - 1;

3.1.3 相关API

检验扩展功能是否存在

入口:

AH = 41h

BX = 55AAh

DL = 驱动器号

返回:

CF = 0

AH = 扩展功能的主版本号

AL = 内部使用

BX = AA55h

CX = API 子集支持位图

CF = 1

AH = 错误码 01h, 无效命令

这个调用检验对特定的驱动器是否存在扩展功能. 如果进位标志置 1 则此驱动器不支

持扩展功能. 如果进位标志为 0, 同时 BX = AA55h, 则存在扩展功能. 此时 CX 的 0 位

表示是否支持第一个子集, 1 位表示是否支持第二个子集.

对于 1.x 版的扩展 Int13H 来说, 主版本号 AH = 1. AL 是副版本号,但这仅限于 BIOS

内部使用, 任何软件不得检查 AL 的值.

扩展读

入口:

AH = 42h

DL = 驱动器号

DS:DI = 磁盘地址数据包(Disk Address Packet)

返回:

CF = 0, AH = 0 成功

CF = 1, AH = 错误码

这个调用将磁盘上的数据读入内存. 如果出现错误, DAP 的 BlockCoun 项中则记录了出错前实际读取的数据块个数.

3.2 LILO需要解决的问题

由于 LILO 可以从硬盘的任何地方加载内核但是它不支持文件系统，所以它需要解决如下的问题：

1、 在启动过称中通过何种方式寻址内核在硬盘上的地址然后将内核加载到内存?

2、 如何突破传统 BIOS 寻址 8G 的问题？

3.3 LILO解决问题的方法

LILO 通过将系统启动过程中需要的文件（内核镜像、根文件系统、配置参数，第二引导模块等）在硬盘上每扇区的地址转换为 BIOS 调用可识别的磁盘地址（CHS，LBA）模式，并将这些地址保存在一个名为 map 的文件中，同时将特需模块的磁盘地址保存在第一引导模块和第二引导模块中，在启动过程中执行第一引导和第二引导模块时，将通过磁盘地址寻址需要的文件。

利用 BIOS 的 INT 13 的扩展系统调用，使用 LBA 硬盘寻址方式，LILO 可以突破内核文件在 8G 以上的硬盘地址中的寻址。

3.4 LILO的组成

LILO 由三部分组成：

引导装载程序：该模块在系统启动的过程中由 BIOS 加载到内存，它将完成加载内核到内存同时将控制权交给内核的任务。

映射文件生产器（The map installer）：编译 LILO 源代码后生成 lilo 可执行文件，运行 make install 命令，该文件会被安装到/sbin/lilo 中，它的作用是：在 linux 中运行它时，将所有属于 LILO 的文件放在系统合适的地方同时记录在启动时需要的数据的地址。

各种文件：该部分包含 LILO 启动时需要的数据。最重要的文件包括 boot loader 和/boot/map 文件，/boot/map 中记录这内核在磁盘上的磁盘地址信息。另一个比较重要的文件是配置文件，通常处于/etc/lilo.conf 处。

3.5 映射文件生成器

映射文件生产器的主要功能是将 LILO 在引导过称中需要的文件（内核映像、根文件系统[initrd]）的磁盘地址保存在/boot/map 文件中，将第二引导模块[second stage]、参数的数据保存在/boot/map 文件中。

每次配置文件的修改，内核的从新编译以及内核在硬盘位置上的移动，都必须运行/sbin/lilo，以便生成正确的/boot/map 文件。

3.5.1 相关文件

和映射文件相关的主要文件：

activate.c

boot.c

bsect.c

cfg.c
common.c
device.c
edit.c
geometry.c
identify.c
lilo.c
map.c
partition.c
probe.c
raid.c
shs2.c
temp.c

3.5.2 函数与数据结构

下面通过跟踪源码分析/boot/map 的生成，了解/boot/map 文件的结构。

3.5.2.1 基本数据结构

common.h

```
 46 ;*/typedef struct {        /*
  47                                        block     0
  48 ;*/       unsigned char sector,track; /* CX
  49                                  sa_sector:    .blkb     1
  50                                  sa_track:     .blkb     1
  51 ;*/       unsigned char device,head; /* DX
  52                                  sa_device:    .blkb     1
  53                                  sa_head:      .blkb     1
  54 ;*/       unsigned char num_sect; /* AL
  55                                  sa_num_sect:      .blkb     1
  56 ;*/} SECTOR_ADDR; /*
  57                                  sa_size:
```

```
58                                  endb
```

SECTOR_ADDR 该数据结构用来表示扇区在硬盘上的绝对地址。

geometry.h

```
87 typedef struct {
88      int device,heads;
89      int cylinders,sectors;
90      int start; /* partition offset */
91      int spb; /* sectors per block */
92      int fd,file;
93      int boot; /* non-zero after geo_open_boot */
94      int raid; /* file references require raid1 relocation */
95      dev_t dev, base_dev; /* real device if remapping (LVM, etc) */
96 } GEOMETRY;
```

GEOMETRY 表示文件所在块设备的物理参数。
device：设备号
heads：磁头数
cylinders：柱面数
start：该设备在整个硬盘的其实扇区
spb：没块数据包含的扇区数

3.5.2.2 基本函数

该函数计算 geo->fd 文件偏移为 offset 对应的磁盘扇区地址，将地址保存在 addr 中

int geo_comp_addr(GEOMETRY *geo,int offset,SECTOR_ADDR *addr)

创建 map 文件

void map_create(char *name)

将 map,last 重置为 NULL

void map_begin_section(void)

将 geo->fd 指定文件的内容写入 map 文件，调用 map_add 将该文件对应的 SECTOR_ADDR 保存到 map 链表中

off_t map_insert_file(GEOMETRY *geo, int skip, int sectors)

将一个扇区的内容写入到 map 文件，同事将该扇区在 map 文件中对应的 SECTOR_ADDR 地址通过 map_register 保存在 map 链表中

void map_add_sector(void *sector)

将 addr 对应的 SECTOR_ADDR 内容保存到 map 链表中

void map_register(SECTOR_ADDR *addr)

将 map 链表中对应的 SECTOR_ADDR 结构体按顺序写入到/boot/map 文件中，并且将 map 链表在/boot/map 中的起始扇区的磁盘地址通过 map_alloc_page 保存在 addr 中。

int map_end_section(SECTOR_ADDR *addr,int dont_compact)

将 map 链表中的 SECTOR_ADDR 元素保存在 SECTOR_ADDR 类型的 list 数组中。

int map_write(SECTOR_ADDR *list,int max_len,int terminate,int sa6)

将 geo->fd 偏移量为 from 开始的 num_sect 个扇区地址装换为扇区磁盘地址，保存在 map 链表的尾部。

void map_add(GEOMETRY *geo,int from,int num_sect)

geometry.c

//geo[in]：文件所在块设备的物理信息

//offset[in]：特定文件（geo->fd）文件的偏移位置

//addr[out]：扇区硬盘地址

//功能：该函数计算 geo->fd 文件偏移为 offset 对应的磁盘扇区

//地址，将地址保存在 addr 中。

```
1353 int geo_comp_addr(GEOMETRY *geo,int offset,SECTOR_ADDR *addr)
1354 {
1355        int block,sector;
```

```
1356        static int linear_warnings = 0;

            //将文件偏移量转换为以块为单位
1370        block = offset/geo->spb/SECTOR_SIZE;

            //将文件偏移对应的块号装换为在块设备中对应的块号
            //此处的 block[in/out]参数，[in]文件偏移块号,[out]磁盘块号
1400        if (ioctl(geo->fd,FIBMAP,&block) < 0) pdie("ioctl FIBMAP");
1401        if (!block) {
1402            return 0;
1403        }

            //将块设备对应的块号装换为扇区号
1439        sector = block*geo->spb+((offset/SECTOR_SIZE) %
geo->spb);

            //将块设备对应的块号装换为绝对扇区号
1473        sector += geo->start;
            …
            //将绝对扇区号装换为磁盘扇区地址
1485        addr->num_sect = linear ? 1 : (sector >> 24);
1486        addr->sector = sector & 0xff;
1487        addr->track = (sector >> 8) & 0xff;
1488        addr->head = sector >> 16;
            …
1534        return 1;
1535 }
```

map.c

```
82 void map_create(char *name)
83 {
            //MAX_DESCR_SECS=3
```

```
104 /* write default command line, descriptor table, zero sector */
105
106         for (i=0; i<MAX_DESCR_SECS+2; i++) {
107         if (write(map_file,buffer,SECTOR_SIZE) != SECTOR_SIZE)
108                 die("write %s: %s",name,strerror(errno));
109         *(unsigned short *) buffer = 0;
110         }
111if(!geo_comp_addr(&map_geo,SECTOR_SIZE*(MAX_DESCR_SEC
S+1),&zero_addr))
112         die("Hole found in map file (zero sector)");
113 }
```

188 void map_begin_section(void)

```
189 {
190         map = last = NULL;
191 }
```

423 off_t map_insert_file(GEOMETRY *geo, int skip, int sectors)

```
424 {
425         off_t here;
426         int count, i;
427         char buff[SECTOR_SIZE];
428
429         if (verbose>0) printf("Calling map_insert_file\n");
430         if (lseek(geo->fd, (off_t)skip*SECTOR_SIZE, SEEK_SET)<0)
431         pdie("map_insert_file: file seek");
432         here = lseek(map_file, 0, SEEK_CUR);
433
434         for (i=0; i<sectors; i++) {
435         count = read(geo->fd, buff, SECTOR_SIZE);
436         if (count<0) pdie("map_insert_file: file read");
```

```
437         if (count<SECTOR_SIZE) memset(buff+count, 0,
SECTOR_SIZE-count);
438         count = write(map_file, buff, SECTOR_SIZE);
439         if (count<=0) pdie("map_insert_file: map write");
440         }
441
442         if ((here % SECTOR_SIZE) != 0) die("Map file positioning
error");
443         map_add(&map_geo, here/SECTOR_SIZE, sectors);
444
445         return here;
446 }
```

174 void map_add_sector(void *sector)

```
175 {
176         int here;
177         SECTOR_ADDR addr;
178
179         if ((here = lseek(map_file,0L,SEEK_CUR)) < 0) pdie("lseek map
file");
180         if (write(map_file,sector,SECTOR_SIZE) != SECTOR_SIZE)
181         pdie("write map file");
182         if (!geo_comp_addr(&map_geo,here,&addr))
183         die("Hole found in map file (app. sector)");
184         map_register(&addr);
185 }
```

161 void map_register(SECTOR_ADDR *addr)

```
162 {
163         MAP_ENTRY *new;
164
165         new = alloc_t(MAP_ENTRY);
166         new->addr = *addr;
```

```
167         new->next = NULL;
168         if (last) last->next = new;
169         else map = new;
170         last = new;
171 }
```

293 int map_end_section(SECTOR_ADDR *addr,int dont_compact)

```
294 {
295         int first,offset,sectors;
296         char buffer[SECTOR_SIZE];
297         MAP_ENTRY *walk,*next;
298         int hinib;
299
300         first = 1;
301         memset(buffer,0,SECTOR_SIZE);
302         offset = sectors = 0;
303         if (compact) map_compact(dont_compact);
304         if (!map) die("Empty map section");
305         hinib = 0;
306         for (walk = map; walk; walk = next) {
307         next = walk->next;
308         if (verbose > 3) {
309                 if ((walk->addr.device&LBA32_FLAG) &&
(walk->addr.device&LBA32_NOCOUNT)) hinib = walk->addr.num_sect;
310                 printf("    Mapped AL=0x%02x CX=0x%04x
DX=0x%04x",walk->addr.num_sect,
311                     (walk->addr.track << 8) |
walk->addr.sector,(walk->addr.head << 8)
312                     | walk->addr.device);
313                 if (linear||lba32)
314                 printf(", %s=%d",
315                     lba32 ? "LBA" : "linear",
```

```
316                 (walk->addr.head << 16) | (walk->addr.track << 8) |
walk->addr.sector | hinib<<24);
317             printf("\n");
318         }
319         if (first) {
320             first = 0;
321             map_alloc_page(0,addr);
322         }
323         if (offset+sizeof(SECTOR_ADDR)*2 > SECTOR_SIZE) {
324             map_alloc_page(SECTOR_SIZE,(SECTOR_ADDR *)
(buffer+offset));
325                 if (write(map_file,buffer,SECTOR_SIZE) !=
SECTOR_SIZE)
326             pdie("write map file");
327             memset(buffer,0,SECTOR_SIZE);
328             offset = 0;
329         }
330         memcpy(buffer+offset,&walk->addr,sizeof(SECTOR_ADDR));
331         offset += sizeof(SECTOR_ADDR);
332         sectors += (walk->addr.device&LBA32_FLAG) &&
(walk->addr.device&LBA32_NOCOUNT)
333             ? 1 : walk->addr.num_sect;
334         free(walk);
335         }
336         if (offset)
337         if (write(map_file,buffer,SECTOR_SIZE) != SECTOR_SIZE)
338             pdie("write map file");
339         return sectors;
340 }
```

```
379 #ifdef LCF_FIRST6
```

```
380 int map_write(SECTOR_ADDR *list,int max_len,int terminate,int sa6)
381 #else
382 int map_write(SECTOR_ADDR *list,int max_len,int terminate)
383 #endif
384 {
385         MAP_ENTRY *walk,*next;
386         int sectors;
387 #ifdef LCF_FIRST6
388         SECTOR_ADDR6 sa6tem, *list6 = (void*)list;
389         unsigned int *list4 = (void*)list;
390 #endif
391
392         sectors = 0;
393         for (walk = map; walk; walk = next) {
394         next = walk->next;
395         if (--max_len < (terminate ? 1 : 0)) die("Map segment is too
big.");
396 #ifdef LCF_FIRST6
397         if (sa6) {
398                 (void)sa6_from_sa(&sa6tem, &(walk->addr));
399                 if (sa6==2) *list4++ = sa6tem.sector;
400                 else *list6++ = sa6tem;
401         }
402         else
403 #endif
404         *list++ = walk->addr;
405
406         free(walk);
407         sectors++;
408         }
409
410         if (terminate) {
```

```
411 #ifdef LCF_FIRST6
412         if (sa6==2) *list4 = 0;
413         else if (sa6) memset(list6, 0, sizeof(SECTOR_ADDR6));
414         else
415 #endif
416         memset(list,0,sizeof(SECTOR_ADDR));
417         }
418
419         return sectors;
420 }
```

```
280 static void map_alloc_page(int offset,SECTOR_ADDR *addr)
281 {
282         int here;
283
284         if ((here = lseek(map_file,offset,SEEK_CUR)) < 0) pdie("lseek
map file")       ;
285         if (write(map_file,"",1) != 1) pdie("write map file");
286         if (fdatasync(map_file)) pdie("fdatasync map file");
287         if (!geo_comp_addr(&map_geo,here,addr))
288         die("Hole found in map file (alloc_page)");
289         if (lseek(map_file,-offset-1,SEEK_CUR) < 0) pdie("lseek map
file");
290 }
```

```
194 void map_add(GEOMETRY *geo,int from,int num_sect)
195 {
196         int count;
197         SECTOR_ADDR addr;
198
199         for (count = 0; count < num_sect; count++) {
```

```
200         if (geo_comp_addr(geo,SECTOR_SIZE*(count+from),&addr))
201                 map_register(&addr);
202         else {
203                 map_register(&zero_addr);
204                 if (verbose > 3) printf("Covering hole at sector
%d.\n",count);
205         }
206         }
207 }
```

```
bsect.c
568 void bsect_open(char *boot_dev,char *map_file,char *install,int delay,
569     int timeout, int raid_offset)
          //….
619         map_create(temp_map);
```

```
82 void map_create(char *name)
83 {
          //MAX_DESCR_SECS=3
104 /* write default command line, descriptor table, zero sector */
105
106         for (i=0; i<MAX_DESCR_SECS+2; i++) {
107         if (write(map_file,buffer,SECTOR_SIZE) != SECTOR_SIZE)
108                 die("write %s: %s",name,strerror(errno));
109         *(unsigned short *) buffer = 0;
110         }
111if(!geo_comp_addr(&map_geo,SECTOR_SIZE*(MAX_DESCR_SEC
S+1),&zero_addr))
112         die("Hole found in map file (zero sector)");
113 }
```

3.5.3 /boot/map的生成

3.5.3.1 map_create

```
82 void map_create(char *name)
83 {
        //MAX_DESCR_SECS=3
104 /* write default command line, descriptor table, zero sector */
105
106     for (i=0; i<MAX_DESCR_SECS+2; i++) {
107     if (write(map_file,buffer,SECTOR_SIZE) != SECTOR_SIZE)
108         die("write %s: %s",name,strerror(errno));
109     *(unsigned short *) buffer = 0;
110     }
111if(!geo_comp_addr(&map_geo,SECTOR_SIZE*(MAX_DESCR_SECS+1),&zero_addr))
112     die("Hole found in map file (zero sector)");
113 }
```

创建 MAP 文件，并将前 5 个扇区作为 缺省参数、描述表、0 扇区。

3.5.3.2 Second stage

第 6 个扇区开始存放 second.b

```
map_begin_section();
here2 = map_insert_data (loader->data, loader->size);
sectors=map_write((SECTOR_ADDR*)secondary_map,(SECTOR_SIZE-4)/sizeof(SECTOR_ADDR)-2, 1, 2);
```

将 map 链表中每个元素的 SECTOR_ADDR 写入到 secondary_map 中。

3.5.3.3 Second stage 对应的SECTOR_ADDR

```
map_begin_section();
map_add_sector(secondary_map);
```

map_register(&addr)

将 addr 表示的扇区的 addr 保存在 bsect 中。在启动的第一阶段，将根据该地址加载启动的第二阶段。

(void) map_write(&bsect.par_1.secondary,1,0,1);

3.5.3.4 fallback,options

map_begin_section();

map_add_sector(fallback_buf);

map_add_sector(options);

3.5.3.5 内核 image

do_image

fd = geo_open(&geo,spec,O_RDONLY);

map_add(&geo,0,(st.st_size+SECTOR_SIZE-1)/SECTOR_SIZE);

//将 map 链表中对应的 SECTOR_ADDR 写入到 map 文件中。

//desc->start 记录内核 SECTOR_ADDR 内容在 map 中的起始地址。

map_end_section(&descr->start,setup+SPECIAL_SECTORS+SPECIAL_BOOTSECT);

geo_close(&geo);

3.5.3.6 文件较大，一个扇区不足以存储文件的所有 SECTOR_ADDR该如何处理？

如果 map 在硬盘上是连续存储的，这不存在这个问题，但是没法保证这一点。因为文件 SECTOR_ADDR 也是存储在硬盘上的，内核在 map 文件中起始扇区的 SECTOR_ADDR 地址保存在 desc->start 中，如果内核足够大，其 SECTOR_ADDR 无法在一个扇区中存储完成，那么下一个扇区的 SECTOR_ADDR 将保存在何处？

在 lilo 的 map_end_section 中有如下的代码：

（1）if (offset+sizeof(SECTOR_ADDR)*2 > SECTOR_SIZE)
（2）{
（3）map_alloc_page(SECTOR_SIZE,(SECTOR_ADDR *) (buffer+offset));
（4）if (write(map_file,buffer,SECTOR_SIZE) != SECTOR_SIZE)
（5）　　pdie("write map file");
（6）memset(buffer,0,SECTOR_SIZE);
（7）offset = 0;
（8）}

第（1）行的判断条件表示的是：

offset+sizeof(SECTOR_ADDR)<= SECTOR_SIZE< offset+sizeof(SECTOR_ADDR)*2

该 SECTOR 的最后几个字节留作它用，保留的字节数 N 满足条件

N>=sizeof(SECTOR_ADDR) && N<2* sizeof(SECTOR_ADDR)

第（2）行调用 map_alloc_page 函数，该函数的代码如下：

```
map_alloc_page(int offset,SECTOR_ADDR *addr)
{
    int here;
    //计算下一个扇区在文件内部的偏移
    if ((here = lseek(map_file,offset,SEEK_CUR)) < 0) pdie("lseek map file");
    if (write(map_file,"",1) != 1) pdie("write map file");
    if (fdatasync(map_file)) pdie("fdatasync map file");
    //将下一个扇区在文件内部的偏移装换成磁盘扇区地址
    if (!geo_comp_addr(&map_geo,here,addr))
        die("Hole found in map file (alloc_page)");
    if (lseek(map_file,-offset-1,SEEK_CUR) < 0) pdie("lseek map file");
}
```

从上面的分析可以看出，每一个扇区的最后一个 SECTOR_ADDR 保存的是下一个存储 SECTOR_ADDR 扇区的 SECTOR_ADDR 地址。通过这个地址，可以将整个内核文件的 SECTOR_ADDR 在内存中保存的扇区位置链接起来。

3.5.3.7 initrd文件

```
fd = geo_open(&geo,initrd,O_RDONLY);
map_begin_section();
map_add(&geo,0,(st.st_size+SECTOR_SIZE-1)/SECTOR_SIZE);
sectors = map_end_section(&descr->initrd,0);
geo_close(&geo);
```

3.5.4 /boot/map文件的内容安排

从 3.5.3 的分析可以看出，map 文件的内容安排如下：

缺省参数、描述表、0 扇区。(1-5 扇区)
second stage(第 6 扇区开始)
second stage 在/boot/map 文件中每 扇区对应的 SECTOR_ADDR 数据
fallback
options
内核映像的 SECTOR_ADDR
根文件系统(INITRD)的 SECTOR_ADDR

从上面的内容可以看出，/boot/map 文件包含两部分内容：

1、 数据区：主要包含缺省参数、描述表、0 扇区，第二引导阶段、fallback、options

2、 内核映像和根文件系统的扇区磁盘地址。

```
henry@henry-desktop:~/lilo/lilo$ sudo hexdump -C /boot/map -s 2048 -n 512
00000800  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000a00
```

/boot/map 第 5 扇区为 0 扇区

```
henry@henry-desktop:~/lilo/lilo$ sudo hexdump -C /boot/map -s 2560 -n 512
00000a00  eb 4e 00 00 00 00 4c 49  4c 4f 16 08 6f f1 cc 4e  |.N....LILO..o..N|
00000a10  02 01 00 00 b6 00 00 00  00 00 04 3d e0 14 02 00  |...........=....|
00000a20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000a30  ff ff 00 00 01 93 00 00  ff ff 00 00 00 93 00 00  |................|
00000a40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000a50  fc 1e 0f a1 2e 8c 1e 02  00 2e 89 16 f6 21 cd 12  |.............!..|
00000a60  e8 28 1c c1 e0 06 2d 60  03 8e c0 0e 1f 31 f6 31  |.(....-`.....1.1|
00000a70  ff 31 c0 b9 00 12 f3 a5  81 c7 00 0e b9 00 01 f3  |.1..............|
00000a80  ab 06 68 86 00 cb e8 27  11 b9 20 00 b4 01 cd 16  |..h....'.. .....|
00000a90  74 06 30 e4 cd 16 e2 f4  e8 f0 1b b0 4c e8 1d 0c  |t.0.........L...|
00000aa0  6a 00 1f c5 36 78 00 80  7c 04 09 77 1e 0e 07 bf  |j...6x..|..w....|
00000ab0  34 22 b9 06 00 f3 a5 26  c6 45 f8 12 6a 00 1f fa  |4".....&.E..j...|
00000ac0  c7 06 78 00 34 22 8c 06  7a 00 fb 2e c6 06 0c 22  |..x.4"..z......"|
00000ad0  00 e8 9e 0c e8 b4 1b 8c  cb 8e db 8e c3 81 eb 00  |................|
00000ae0  04 b9 00 90 39 cb 76 02  89 cb c7 06 fc 21 00 26  |....9.v......!.&|
00000af0  89 1e 6e 0b 8d 4f 20 89  0e 6c 0b 8c c9 29 d9 c1  |..n..O ..l...)..|
00000b00  e1 04 8e d3 89 cc 60 bb  39 1f e8 a2 0b a1 6e 0b  |......`.9.....n.|
00000b10  e8 65 0b a1 6c 0b e8 58  0b 8c c8 e8 53 0b 8c d0  |.e..l..X....S...|
00000b20  e8 4e 0b b0 3a e8 95 0b  89 e0 e8 4b 0b b0 20 e8  |.N..:......K.. .|
00000b30  8b 0b e8 56 1b 64 a1 fe  07 e8 35 0b a1 f6 21 e8  |...V.d....5...!.|
00000b40  2f 0b e8 09 00 20 66 6c  61 67 73 32 3d 00 5b e8  |/.... flags2=.[.|
00000b50  5d 0b a0 1f 00 e8 27 0b  e8 5b 0b e8 59 00 52 65  |].....'..[..Y.Re|
00000b60  67 69 73 74 65 72 73 20  61 74 20 73 74 61 72 74  |gisters at start|
00000b70  75 70 20 6f 66 20 66 69  72 73 74 20 73 74 61 67  |up of first stag|
00000b80  65 20 6c 6f 61 64 65 72  3a 0a 20 41 58 20 20 20  |e loader:. AX   |
00000b90  42 58 20 20 20 43 58 20  20 20 44 58 20 20 20 53  |BX   CX   DX   S|
00000ba0  49 20 20 20 44 49 20 20  20 42 50 20 20 20 44 53  |I   DI   BP   DS|
00000bb0  20 20 20 45 53 0a 00 5b  e8 f4 0a e8 cd 1a 64 a1  |   ES..[......d.|
00000bc0  f2 07 e8 b3 0a 64 a1 ec  07 e8 a5 0a 64 a1 f0 07  |.....d......d...|
00000bd0  e8 9e 0a 64 a1 ee 07 e8  97 0a 64 a1 e6 07 e8 90  |...d......d.....|
00000be0  0a 64 a1 e4 07 e8 89 0a  64 a1 e8 07 e8 82 0a 64  |.d......d......d|
00000bf0  a1 f6 07 e8 7b 0a 64 a1  f4 07 e8 74 0a e8 b6 0a  |....{.d....t....|
```

/boot/map 第 6 扇区内容

```
henry@henry-desktop:~/lilo/lilo$ sudo hexdump -C second.b  -n 512
00000000  eb 4e 00 00 00 00 4c 49  4c 4f 16 08 00 00 00 00  |.N....LILO......|
00000010  02 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  ff ff 00 00 01 93 00 00  ff ff 00 00 00 93 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  fc 1e 0f a1 2e 8c 1e 02  00 2e 89 16 f6 21 cd 12  |.............!..|
00000060  e8 28 1c c1 e0 06 2d 60  03 8e c0 0e 1f 31 f6 31  |.(....-`.....1.1|
00000070  ff 31 c0 b9 00 12 f3 a5  81 c7 00 0e b9 00 01 f3  |.1..............|
00000080  ab 06 68 86 00 cb e8 27  11 b9 20 00 b4 01 cd 16  |..h....'.. .....|
00000090  74 06 30 e4 cd 16 e2 f4  e8 f0 1b b0 4c e8 1d 0c  |t.0.........L...|
000000a0  6a 00 1f c5 36 78 00 80  7c 04 09 77 1e 0e 07 bf  |j...6x..|..w....|
000000b0  34 22 b9 06 00 f3 a5 26  c6 45 f8 12 6a 00 1f fa  |4".....&.E..j...|
000000c0  c7 06 78 00 34 22 8c 06  7a 00 fb 2e c6 06 0c 22  |..x.4"..z......"|
000000d0  00 e8 9e 0c e8 b4 1b 8c  cb 8e db 8e c3 81 eb 00  |................|
000000e0  04 b9 00 90 39 cb 76 02  89 cb c7 06 fc 21 00 26  |....9.v......!.&|
000000f0  89 1e 6e 0b 8d 4f 20 89  0e 6c 0b 8c c9 29 d9 c1  |..n..O ..l...)..|
00000100  e1 04 8e d3 89 cc 60 bb  39 1f e8 a2 0b a1 6e 0b  |......`.9.....n.|
00000110  e8 65 0b a1 6c 0b e8 58  0b 8c c8 e8 53 0b 8c d0  |.e..l..X....S...|
00000120  e8 4e 0b b0 3a e8 95 0b  89 e0 e8 4b 0b b0 20 e8  |.N..:......K.. .|
00000130  8b 0b e8 56 1b 64 a1 fe  07 e8 35 0b a1 f6 21 e8  |...V.d....5...!.|
00000140  2f 0b e8 09 00 20 66 6c  61 67 73 32 3d 00 5b e8  |/.... flags2=.[.|
00000150  5d 0b a0 1f 00 e8 27 0b  e8 5b 0b e8 59 00 52 65  |].....'..[..Y.Re|
00000160  67 69 73 74 65 72 73 20  61 74 20 73 74 61 72 74  |gisters at start|
00000170  75 70 20 6f 66 20 66 69  72 73 74 20 73 74 61 67  |up of first stag|
00000180  65 20 6c 6f 61 64 65 72  3a 0a 20 41 58 20 20 20  |e loader:. AX   |
00000190  42 58 20 20 20 43 58 20  20 20 44 58 20 20 20 53  |BX   CX   DX   S|
000001a0  49 20 20 20 44 49 20 20  20 42 50 20 20 20 44 53  |I   DI   BP   DS|
000001b0  20 20 20 45 53 0a 00 5b  e8 f4 0a e8 cd 1a 64 a1  |   ES..[......d.|
000001c0  f2 07 e8 b3 0a 64 a1 ec  07 e8 a5 0a 64 a1 f0 07  |.....d......d...|
000001d0  e8 9e 0a 64 a1 ee 07 e8  97 0a 64 a1 e6 07 e8 90  |...d......d.....|
000001e0  0a 64 a1 e4 07 e8 89 0a  64 a1 e8 07 e8 82 0a 64  |.d......d......d|
000001f0  a1 f6 07 e8 7b 0a 64 a1  f4 07 e8 74 0a e8 b6 0a  |....{.d....t....|
```

second.b 第一个扇区的内容

3.5.5 总结

从前面的分析可以看出，map 文件保存了一些重要的信息，如 LILO 装载程序的第二引导部分（second stage），第二引导部分每扇区对应的 SECTOR_ADDR 数据，内核镜像文件的 SECTOR_ADDR 数据，根文件系统的（如果有的话）SECTOR_ADDR 数据。

LILO 引导过称中，必须读取 LILO 装载程序的第二引导部分，读取内核镜像文件的 SECTOR_ADDR 数据，根文件系统的（如果有的话）SECTOR_ADDR 数据，然后根据这些信息来加载内核镜像和根文件系统。那么“LILO 装载程序的第二引导部分”在/boot/map 文件中每个扇区对应的 SECTOR_ADDR 地址是如何传入引导程序的？“内核镜像文件的 SECTOR_ADDR 数据”，“ 根文件系统的（如果有的话）SECTOR_ADDR 数据”在/boot/map 文件中每个扇区对应的 SECTOR_ADDR 又是如何传入引导程序的？

让我们带着上面的疑问，进入下一小节**引导装载程序。**

3.6 引导装载程序

引导装载程序由第一引导阶段（first stage）和第二引导阶段（second stage）两部分组成：

first stage：first stage 的会写入 MBR 或者引导扇区。主要功能是根据保存在/boot/map 文件中的 second stage 的 SECTOR_ADDR 地址，将 second stage 读入内存合适的地方，然后将控制权移交给 second stage。

second stage：通过保存在/boot/map 文件中的 SECTOR_ADDR 地址，将内核镜像和根文件系统（如果存在的话）加载到内核，然后将控制权移交给内核的 setup 部分（0x9020:0000）。

3.6.1 相关文件

biosdata.S
bitmap.S
bootsect.S
chain.S
crt.S
disk.S
display4.S
dparam.S
dump.S
first.S
graph.S
mapper.S
mbr.S
menu.S
pseudo.S
read.S
second.S
shs3.S
strlen.S
volume.S

其中重点关注 first.S，second.S，它们分别代表了 first stage 和 second stage。其它的文件都是为它们服务的。

3.6.2 first.S

第一引导阶段，编译后生产 512 字节，根据用户的配置，将被写入 MBR 或者是引导扇区。

3.6.2.1 函数

```
preaed 函数：读取一个扇区的内容到 es:bx 指定的缓冲区
508 pread:
509         lodsd                   ! get address
    509 行：ds:si=>ax
510         or    eax,eax
511         jz    done
512         add eax,[raid](bp)    ! reloc is 0 on non-raid
513         call        disk_read
514
515         add bh,#SECTOR_SIZE/256         ! next sector
515 行：bx = bx + 512，自动移动缓冲区起始地址，用于接收下一个扇区
516 done:
517         ret
```

3.6.2.2 源码剖析

当 LILO 作为引导程序时，BIOS 会将 first stage 的内容读入 0x07c0:0000 出，并跳转到_main 出开始执行：

下面分析其源码：

```
60          .text
 61
 62          .globl    _main
 63
 64          .org       0
 65
```

```
66 zero:
67 _main:    cli                ! NT 4 blows up if this is missing
68          jmp start
```

67-68：关中断，跳转到 start 开始执行。

```
130 start:
131         mov ax,#BOOTSEG ! use DS,ES,SS = 0x07C0
132 /***************************************************/
133
134         mov ss,ax
135         mov sp,#SETUP_STACKSIZE ! set the stack for First Stage
136         sti                ! now it is safe
……
192 lagain:
193         pusha                  ! preserve all the registers for restart
194
195         push       ds
196         pop es           ! use buffer at end of boot sector
```

134-196：设置段寄存器 DS，ES，SS 为 0x07c0，设置 video 显示模式，在屏幕上显示”L”。

```
197
198         cmp dl,#EX_DL_MAG      ! possible boot command line (chain.S)
199         jne boot_in_dl
200         mov dl,dh              ! code passed in DH instead
201 boot_in_dl:
202
203         mov bx,#map2       ! buffer for volume search
```

ES:BX = 0x7c00:0200

```
204         mov dh,[d_dev](bp)    ! map device to DH
205
206 #if VALIDATE
```

```
207         mov ax,dx           ! copy device code to AL
208         and ah,#0x80        ! AH = 00 or 80
209         xor al,ah           ! hi-bits must be the same
210         js  use_installed
211         cmp al,#MAX_BIOS_DEVICES    ! limit the device code
212         jae use_installed   ! jump if DL is not valid
213 #endif
214
215 ! map is on boot device for RAID1, and if so marked; viz.,
216
217         test    byte ptr [prompt](bp),#FLAG_MAP_ON_BOOT
218         jnz use_boot        ! as passed in from BIOS or MBR loader
```

这里只分析 LILO 作为启动阶段的代码

```
            262 use_boot:
263         push    bx          ! save map2 for later
```

263 行：保存 bx 的地址。

```
264
265         mov dh,[d_flag](bp) ! get device flags to DH
266         mov si,#d_addr
```

266 行：这一行很重要，它是第一引导阶段在/boot/map 中寻址第二引导阶段并将其加载到内存的关键。

```
267         call    pread           ! increments BX
```

267 行：将 second stage 在/boot/map 文件中每扇区对应的 SECTOR_ADDR 数据读入到 es:bx 指定的缓冲区中。

```
273         pop si          ! point at #map2
```

273 行：si=bx，即 ds:si 指向 267 行保存 pread 进来的 SECTOR_ADD 数据的缓冲区地址

```
274
275 #if 1
276         push    #SETUP_STACKSIZE/16 + BOOTSEG +
SECTOR_SIZE/16*2
277         pop es
```

```
278 #else
279         mov ax,ds           ! get segment
280         add ax,#SETUP_STACKSIZE/16      !     +
SECTOR_SIZE/16*2
281         mov es,ax
282 #endif
283         xor bx,bx
284
285 sload:
286         call      pread           ! read using map at DS:SI
```

286 行：通过 DS:SI 指向的 SECTOR_ADDR 缓冲区地址，根据里面的每一个 SECTOR_ADDR，读取一个扇区

```
287         jnz sload           ! into memory at ES:BX (auto increment)
289 ! Verify second stage loader signature
290
291         mov si,#sig         ! pointer to signature area
292         mov di,si
293         mov cx,#length    ! number of bytes to compare
294         mov ah,#0x9A       ! possible error code
295         repe
296            cmpsb              ! check Signature 1 & 2
297         jne error      ! check Signature 2
```

291-298：验证读入的 second stage 是否正确。

```
298
299 #if SECOND_CHECK
300 /* it would be nice to re-incorporate this check */
301         mov al,#STAGE_SECOND       ! do not touch AH (error code)
302         scasb
303         jne error
304 #endif
305
306 ! Start the second stage loader        DS=location of Params
```

```
307
308     push    es          ! segment of second stage
309     push    bp          ! BP==0
```

308-309：将 es，bp 入栈，为 retf 做准备

```
310
311     mov al,#0x49        ! display an 'I'
312     call    display
313
314     retf                ! Jump to ES:BP
```

314 行：jump to es:bp，即第二引导阶段在内存中的首地址。

3.6.2.3 疑问解析

问题一：在 first.S 的 266 行有代码

266 mov si,#d_addr

当时说#d_addr 对应的内存地址中保存/boot/map 中 second stage 对应的 SECTOR_ADDR 数据的扇区磁盘地址，那么该地址是何时写入 first.S 中的呢？

这个问题的解答得从映射文件生成器执行时说起。

先看看相关的数据结构

Common.h

```
304 typedef union {
305     BOOT_PARAMS_1 par_1;
306     BOOT_PARAMS_2 par_2;
307     BOOT_PARAMS_C par_c;
308     BOOT_PARAMS_DOS par_d;
309     BOOT_VOLID boot;
310     unsigned char sector[SECTOR_SIZE];
311 } BOOT_SECTOR;
```

BOOT_SECTOR 表示启动扇区的数据结构，当其表示 first stage 时，表示的数据类型本质上是一个 BOOT_PARAMS_1

```
131 ;*/typedef struct {                    /*
```

```
132                                     block   0
133 ;*/     unsigned char cli;      /* clear interrupt flag instruction
134                                 par1_cli:   .blkb   1
135 ;*/     unsigned char jmp0, jmp1;       /* short jump
136                                 par1_jump:  .blkb   2
137 ;*/     unsigned char stage;    /*
138                                 par1_stage: .blkb   1
139 ;*/     unsigned short code_length;    /* length of the first stage code
140                                 par1_code_len:  .blkb   2
141 ;*/     char signature[4]; /* "LILO"
142                                 par1_signature: .blkb   4
143 ;*/     unsigned short version;    /*
144                                 par1_version:   .blkb   2
145 ;*/     unsigned int map_stamp; /* timestamp for this installation (map
creation)
146                                 par1_mapstamp:  .blkb   4
147 ;*/     unsigned int raid_offset; /* raid partition/partition offset
148                                 par1_raid_offset: .blkb 4
149 ;*/     unsigned int timestamp; /* timestamp for restoration
150                                 par1_timestamp: .blkb   4
151 ;*/     unsigned int map_serial_no; /* volume serial no. / id containing
the map file
152                                 par1_map_serial_no: .blkb   4
153 ;*/     unsigned short prompt; /* FLAG_PROMPT=always,
FLAG_RAID install
154                                 par1_prompt:    .blkb   2
155 ;*/     SECTOR_ADDR secondary; /* sectors of the second stage
loader
156                                 par1_secondary: .blkb   sa_size+1
157 ;*/} BOOT_PARAMS_1; /* first stage boot loader
158                                     .align  4
159                                 par1_size:
```

可以看到 BOOT_PARAMS_1 中元素的排列顺序和 first.S 中的一致

但有一处需要注意的地方是：

first.S 中

```
91 prompt: .word    0          ! indicates whether to always enter prompt
92                             ! contains many other flags, too
93
94 d_dev:   .byte    0x80           ! map file device code
95 d_flag: .byte    0          ! disk addressing flags
96 d_addr: .long    0          ! disk addr of second stage index sector
```

Prompt 之后有 6 个字节，即 d_dev 1 字节，d_flag 1 字节，a_addr 4 字节。

而 BOOT_PARAMS_1 中确只用了 5 字节的 SECTOR_ADDR 数据结构来与之对应，不过

```
par1_secondary: .blkb    sa_size+1
```

可以影响其大小，使其变为 6 字节。这个是编译器内部实现的，具体原理还没有搞懂。sizeof(BOOT_PARAMS_1) = 36，表示编译器确实为 secondary 分配了 6 个字节的空间。

在运行/sbin/lilo 重新安装 LILO 引导程序时，lilo 会根据 second stage 在 /boot/map 上的具体位置，重置 first stage 中的 d_addr 字段，让其指向正确的位置。这也是为啥每次更新 LILO 相关的内容时多必须重新运行/sbin/lilo 的缘故。

具体内容看源码：

```
static BOOT_SECTOR bsect,bsect_orig;
```

lilo.c

```
556 int main(int argc,char **argv)
{
  //调用 bsect_open
968     bsect_open(
969         cfg_get_strg(cf_options,"boot"),
970         cfg_get_strg(cf_options,"map") ?
971             cfg_get_strg(cf_options,"map") : MAP_FILE,
972         cfg_get_strg(cf_options,"install"),
```

```
973             cfg_get_strg(cf_options,"delay") ?
974                 timer_number(cfg_get_strg(cf_options,"delay")) : 0,
975             cfg_get_strg(cf_options,"timeout") ?
976                 timer_number(cfg_get_strg(cf_options,"timeout")) : -1,
977             raid_offset );
    }
```

bsect_open.c

```
568 void bsect_open(char *boot_dev,char *map_file,char *install,int delay,
569     int timeout, int raid_offset)
{
//将用户指定(boot_dev)的块设备的引导扇区的内容读入到
bsect,bsect_orig 中。
open_bsect(boot_dev);
//创建 map_create 文件
619         map_create(temp_map);
620         temp_register(temp_map);
//根据用户的选择，保存第二引导阶段的内容到 loader 中。
loader = select_loader();
//将编译生成的第一引导阶段的内容拷贝到 bsect 中。
memcpy(&bsect, First.data, MAX_BOOT_SIZE);
//将第二引导阶段的内容写入到/boot/map 文件中。
647         here2 = map_insert_data (loader->data, loader->size);
 648         memcpy(&param2,loader->data,sizeof(param2));
 649 #ifdef LCF_FIRST6
 650         /* write just the 4 bytes (sa6==2) */
//将第二引导阶段对应的 SECTOR_ADDR 写入到 secondary_map 数组中
 651         sectors = map_write((SECTOR_ADDR*)secondary_map,
(SECTOR_SIZE-4)/sizeof(SECTOR_ADDR)-2, 1, 2);
 652 #else
 653         sectors = map_write((SECTOR_ADDR*)secondary_map,
(SECTOR_SIZE-4)/sizeof(SECTOR_ADDR)-2, 1);
```

```
654 #endif
655         memcpy(secondary_map+SECTOR_SIZE-4,
EX_MAG_STRING, 4);
    //将 secondary_map 中的内容（表示 second stage 的 SECTOR_ADDR 数据）写入到/boot/map 文件中。
    682         map_begin_section();
     683         map_add_sector(secondary_map);
     684 #ifdef LCF_FIRST6
     685         /* write out six byte address */
    //这里，将 secondary_map 在/boot/map 上的 SECTOR_ADDR 地址保存到 bsect.par_1.secondary 中，即保存在第一引导阶段的 d_addr 变量中。这里是 LILO 一切动作的入口点。
     686         (void) map_write(&bsect.par_1.secondary,1,0,1);
     687 #else
     688         (void) map_write(&bsect.par_1.secondary,1,0);
     689 #endif


    }
```

当返回到 main.c 中时，会根据用户的指定的参数执行 bsect_update 函数，将 bsect 写入到指定的引导扇区中。

lilo.c

```
1003            cp = cfg_get_strg(cf_options,"force-backup");
1004            if (cp) bsect_update(cp,1,0);
1005            else bsect_update(cfg_get_strg(cf_options,"backup"),0,0);
```

在 first.S 的 266 行有代码

问题二：根据问题一中的描述，第二引导阶段生成的二进制文件最大为多少字节？

在 bsect.c 中有如下的定义

static char secondary_map[SECTOR_SIZE];

也就是说在 LILO 中只用了一个扇区（512 字节）的大小保存第二引导阶段对应的 SECTOR_ADDR 数据，而每个 SECTOR_ADDR 占 5 字节，所以

secondary_map 可以保存 512/5=102 个 SECTOR_ADDR 数据，而每个 SECTOR_ADDR 对应第二引导阶段的一个扇区，所以 LILO 第二引导阶段生成的二进制文件最大可以为 102*0.5k=51k，这个大小对于第 2 引导阶段已经足够了，在 LILO 22.8 中的第二引导阶段的大小为 9k 左右。

3.6.2.4 内存布局

0x07c00-0x07dff	512 bytes 第一引导阶段
0x07e00-0x080ff	512 bytes 第二引导阶段的 SECTOR_ADDR
0x08800-0x0ac00	9k second stage 第二引导阶段

3.6.3 second.S

第一引导阶段将第二引导阶段加载到内存后，开始执行第二引导阶段的代码。
第二引导阶段主要做如下几件事：

1、 将自己迁移到合适的内存
2、 加载 keytab，descr，default paramter 到内存
3、 加载内核内核的引导扇区代码到 0x9000:0000 出
4、 加载内核的 setup 代码到 0x9020:0000 出。
5、 加载内核到 0x100000 处。
6、 加载 initrd（如果有的话到合适的内存）。
7、 跳转到 0x9020:0000，将控制权交给内核。

3.6.3.1 关键代码剖析

second.S

```
117         .text
 118
 119         .globl    _main
 120         .org      0
 121
 122 _main:    jmp start
```

122 行：跳转到 start 开始执行。

```
182 start:    cld                ; only CLD in the code; there is no STD
183 #if     ! NO_FS
 184        push       ds
 185        pop fs           ; address parameters from here
 186 #endif
 187 #if NO_FS || DEBUG_NEW
 188        seg cs
 189        mov firstseg,ds ; save DS here
 190 #endif
 191
 192        seg cs
 193        mov [init_dx],dx       ; save DX passed in from first.S
 194
 195        int 0x12               ; get memory available
 196        CHECK_FS
 197 #if EBDA_EXTRA
 198        sub ax,#EBDA_EXTRA    ; allocate extra EBDA
 199 #endif
 200        shl ax,#6              ; convert to paragraphs
 201        sub ax,#Dataend/16
 202        mov es,ax              ; destination address
 203        push       cs
 204        pop ds
 205        xor si,si
 206        xor di,di
 207        xor ax,ax
 208        mov cx,#max_secondary/2 ; count of words to move
 209        rep
 210           movsw
```

209-210 行：将 second stage 转移到合适的内存

```
 211        add di,#BSSstart-max_secondary
```

```
212     mov cx,#BSSsize/2
213     rep
214       stosw
215     push    es
216     push    #continue
217     retf                ; branch to continue address
```

215-217 行：跳转到转移后的 continue 继续执行。

```
218 continue:


236 drkbd:  mov ah,#1           ; is a key pressed ?
237     int 0x16
238     jz  comcom          ; no -> done
239     xor ah,ah           ; get the key
240     int 0x16
241     loop    drkbd
242 #endif
243
244 comcom:
245     CHECK_FS
246     mov al,#0x4c        ; display an 'L'
247     call    display
```

246-247 行：显示”LILO”中的第二个 L.

```
294 restrt1:
295     mov word ptr [map],#MAP
296     mov [initseg],bx    ; set up INITSEG (was 0x9000)
297     lea cx,(bx+0x20)
298     mov [setupseg],cx   ; set up SETUPSEG (was 0x9020)
299     mov cx,cs
300     sub cx,bx           ; subtract [initseg]
301     shl cx,#4           ; get stack size
302     mov ss,bx           ; must lock with move to SP below
```

```
303         mov sp,cx           ; data on the stack)


                429 ldsc:
430         BEG_FS
431         SEG_FS
432         mov eax,[par1_mapstamp]
433         END_FS
434         cmp eax,[par2_mapstamp]
435         jne timeerr
436
437         call      kt_read       ; read the KEYTABLE
438
439         call      build_vol_tab
440
441         mov bx,#DESCR
442         mov si,#KEYTABLE+256+mt_descr
443 descr_more:
444         lodsw
445         xchg      cx,ax
446         lodsw
447         xchg      dx,ax
448         lodsb
449         call      cread
450         jc   near fdnok   ; error -> retry
451         add bh,#2           ; increment address
452         cmp
si,#KEYTABLE+256+mt_descr+sa_size*MAX_DESCR_SECS_asm
453         jb   descr_more
454
455         mov si,#DESCR     ; compute a checksum of the descriptor
table
456         mov di,#SECTOR_SIZE*MAX_DESCR_SECS-4
```

```
457
458         push       dword #CRC_POLY1
459         call       crc32
460         add di,si
461         cmp eax,dword (di)
462         jz    nochkerr

482 nochkerr:
483 #ifdef DEBUG
484         pusha
485         mov bx,#nochker_msg
486         call       say
487         popa
488         jmp nochkerr1
489 nochker_msg:
490         .ascii    "Descriptor checksum okay\n"
491         .byte     0
492 nochkerr1:
493 #endif
494 #ifdef LCF_VIRTUAL
495 ; remove those items that have "vmdisable", if virtual boot
496         call       vmtest
497         jnc virtual_done              ;jmp

521 virtual_done:
522 #endif
523 #ifdef LCF_NOKEYBOARD
524 ; remove those items that have "nokbdisable", if nokeyboard boot
525         call       kbtest
526         jc    kbd_done              ;jmp
```

```
550 kbd_done:


597         jmp dokay           ; continue


649 dokay:    mov bx,#ospc       ; display 'O '
650         call       say
//649-650：显示 “LILO”的最后一个 O，表示 LILO 所有部分都加载完成。
680 skip_prompt:
 681         mov nodfl,#bfirst     ; boot first image if falling through
 682         call       waitsh          ; wait for a shifting key
 683         jc    iloop           ; key pressed -> enter interactive mode
 684
 685 ! Check for external parameters
 686
 687 extp:     BEG_FS
 688         SEG_FS          ; external parameters ?
 689         cmp byte ptr EX_OFF+6,#EX_DL_MAG
 690         END_FS
 691         jne noex             ; no -> go on
 692         BEG_FS
 693         SEG_FS
 694         mov bl,EX_OFF+7 ; get drive
 695         SEG_FS          ; clear flag
 696         mov byte ptr EX_OFF+6,bl       ; clear flag
 697         SEG_FS          ; load the signature pointer
 698         les bx,EX_OFF
 699         END_FS
 700         seg es
 701         cmp dword ptr (bx),#EX_MAG_HL     ; "LILO"
 702         jne noex             ; no -> go on
```

```
714 noex:    push    cs          ; restore ES
 715         pop es
 716         mov si,#DFLCMD+2      ; default command line ?
 717         cmp byte ptr (si),#0
 718         jne niloop          ; yes -> use it
 719         mov ax,nodfl        ; no idea how to tell as86 to do jmp (addr) :-(
 720         jmp ax          ; fall through
```

//719-720：跳转到 bfirst 开始执行：

```
1049 bfirst: mov byte ptr lkcbuf,#0   ; clear default
1050         cmp byte ptr cmdline,#0 ; is there a default ?
1051         jne bcmd             ; yes -> boot that image
1052 brfrst:
1053         mov bx,#DESCR0   ; boot the first image
```

//1053 行：bx 中保存变量 Descr 的地址。

```
1425 doboot: mov byte ptr prechr,#61 ; switch to equal sign
1426         push    bx          ; save image descr
1427         mov bx,#msg_l     ; say hi
1428         call    say
1429         pop bx          ; display the image name
1430         push    bx
1431         call    say
1432         pop si
```

1432 行：bx=>si，si 指向 image descr

```
1433
1434         push    si
1435         add si,#id_start      ; form address
```

1435 行：该行非常重要，si 指向在 id_start 中保存的内核文件的 SECTOR_ADDR 数据的起始扇区的磁盘地址

```
1436
```

```
1437 ;   Now load the kernel sectors
1438        xor ax,ax
1439        mov word ptr (gdt+0x1b),ax ; set GDT to "load low"
1440        mov byte ptr (gdt+0x1f),al
1441        mov moff,ax         ; map is not loaded yet
1442
1443        lodsw               ; address of the first map sector
```

1443 行：ds:si=>eax

```
1444        xchg      cx,ax
1445        lodsw
1446        xchg      dx,ax
1447              lodsb
1448
1449        push      si        ; save SI
1450
1451 #ifdef DEBUG
1452              push      ax                ;
1453        mov bx,#step0
1454        call say
1455              pop       ax                ;
1456 #endif
1457        mov bx,[map]              ; load the first map sector
1458        call      sread
```

1457-1458 行：将内核 first map sector 读入[map]所指向的内存中。

```
1459 #ifdef DEBUG
1460        mov bx,#step0b
1461        call say
1462 #endif
1463        mov bx,#DFLCMD    ; load the default command line
1464 ;BEG_FS
```

```
1568 cpdone:
1569
1570
1571 #if DEBUG_NEW
1572         push       cx
1573         mov bx,#msg_pl    ; parameter line message
1574         call       say
1575         pop cx
1576         mov ax,#CL_LENGTH-1
1577         sub ax,cx
1578         call       wout
1579         call       crlf
1580 #endif
1581
1582 #ifdef DEBUG
1583         mov bx,#step2
1584         call say
1585 #endif
1586         mov es,[initseg]       ; load the original boot sector
1587         xor bx,bx              ; load now
1588         call       load1
```

1586-1588：加载内核的 原始引导扇区到 0x9000:0000 处。

```
        1621 nolock:
1622 #ifdef DEBUG
1623         mov bx,#step3
1624         call say
1625 #endif
1626         xor cx,cx
1627         seg es
```

```
1628            add     cl,[VSS_NUM]
1629 ;;; or    cx,cx
1630        jnz lsetup
1631        mov cl,#SETUPSECS     ; default is to load four sectors
1632 lsetup:
1633        mov es,[setupseg]     ; load the setup codes
```

1633 行：加载 setup code 到内核 0x9020:0000 开始的内存

```
1634
1635 #ifdef MEMORY_CHECK
1636        mov ax,cx             ; number of sectors to AX
1637        shl ax,#5             ; convert to paragraphs (9-4)
1638        mov bx,es
1639        add bx,ax
1640        add bx,#STACK>>4       ; allow for stack space in paragraphs
1641        mov ax,cs             ;
1642        cmp bx,ax
1643        jbe enough_mem
1644        mov bx,#msg_mem ; we are very short on memory
1645        call       say
1646
1647 enough_mem:
1648 #endif
1649
1650        xor bx,bx             ; other operating system)
1651 lsloop: push       cx
1652        call       loadopt
1653        pop cx
1654        loop       lsloop
```

1651-1654 行：循环加载内核的 setup code

```
        1694        call       load_initrd ; load the initrd & patch header
```

1694 行：加载 load_initrd 代码到内核。（这段代码比较难懂，下一节单独讲解。）

```
1714        call    lfile           ; load the system ...
```

1714 行：加载内核剩余的部分

```
1715        jmp launch2         ; ... and run it

1724 launch2:
1725
1726        jmp launch          ; go !

1790 launch:
```

一系列的检查后，通过 2032 行跳转到 0x9020:0000，将控制权交给内核。

```
2032        jmpi    0,SETUPSEG          ; segment part is a variable
```

3.6.3.2 内核文件的SECTOR_ADDR数据超过一个扇区时，LILO是如何处理的？

在 3.5.3.6 小结中，曾提到问题：文件较大，一个扇区不足以存储文件的所有 SECTOR_ADDR 数据时该如何处理？

内核镜像和 initrd 文件的 SECTOR_ADDR 数据都不止一个扇区，所以加载代码必须能够处理这种情况。下面通过 load_initrd 函数来分析 LILO 是如何处理的。

```
3119 load_initrd:

3134        add si,#id_rd_size    ; point at ramdisk size long
3135 ! take care of the RAM disk first
3136        xor eax,eax
3137        mov (rdbeg),eax ; clear address
3138        lodsd
```

```
3139        mov (rdszl),eax ; set rdszl+rdszh
3140        add eax,#4095     ; round up &
3141        shr eax,#12       ; convert to pages
3142        xchg      bx,ax          ; copy to BX
3143        lodsw                    ; address of the first map sector
```

3143 行：eax 中保存着 initrd 的 map sector 第一个扇区的 磁盘扇区地址

```
3144        xchg      cx,ax
3145        lodsw
3146        xchg      dx,ax
3147              lodsb
3148        or    bx,bx          ; no RAM disk ?
3149        jz    noramd          ; yes -> skip it 2
3150
3151        push      si         ; save SI, ES, and BX (RD size)
3152        push      es
3153        push      bx
3154        mov bx,[map]             ; load the first map sector
3155        call      sread
```

3154-3155 行：读取第一个 map sector 保存在 map 指向的内存块中。

```
    3186        call      lfile          ; load it
```

3186 行：通过 lfile 开始加载 initrd 文件。

```
1732 lfile:   call      load
1733        jmp lfile
```

1732-1733：循环加载 initrd 文件。

```
1753 load:    push      es         ; save ES:BX
1754        push      bx
1755 lfetch: mov si,moff        ; get map offset
1756        mov bx,[map]
```

```
1757        mov cx,(bx+si)    ; get address
1758        mov dx,(bx+si+2)
1759        mov al,(bx+si+4)
```

1756-1759 行：根据加载进来的 map sector（SECTOR_ADDR 数据），加载其指定的每一个扇区

```
1760        or    cx,cx           ; at EOF ?
1761        jnz noteof          ; no -> go on
1762        or    dx,dx
1763        jnz noteof
```

1760-1763 行：所有数据都加载完成吗？是：不跳转，否则跳转到 noteof。

```
1764        pop bx          ; restore ES:BX
1765        pop es
1766        pop ax          ; pop return address
1767        ret             ; return to outer function
```

1766-1767：将返回到 lfile 出的地址给 drop 掉，返回到 3186 行，这样才可以打破 1732-1733 的死循环。

```
1768 noteof: add si,#sa_size ; increment pointer
```

1768 行：si 表示指向 map sector 的指针，每次要递增 sizeof(SECTOR_ADDR)个字节：

```
1769        mov moff,si
1770        cmp si,#SECTOR_SIZE - sa_size + 1           ; page end ?
```

1770 行：判断 si 是否指向 map sector 的最后一个 SECTOR_ADDR 数据，根据之前的讨论，

如果是最后一个，则其表示的扇区是一个 map sector，否则其表示的扇区是 initrd 数据。

```
1771        jb    near doload
```

1771 行：如果不是最后一个，则跳转到 doload 加载 initrd 文件数据,然后返回 lfile 处。如果是最后一个，则加载 map sector 扇区，同时清理 moff。

```
1772
1773        mov moff,#0         ; reset pointer
1774        push      cs        ; adjust ES
1775        pop es
```

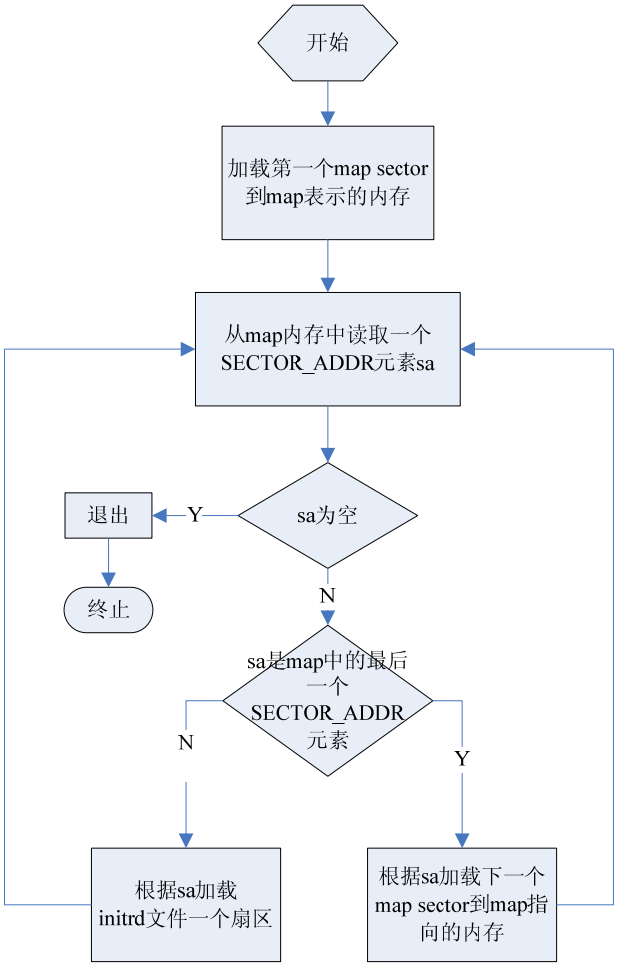
```
1776
1777            mov     bl,hinib            ; this might get clobbered
1778            push    bx                  ; so save it
1779        mov bx,[map]            ; load map page
1780        call    sread
```

1779-1780 行：加载下一个 map sector 到 map 所指向的内存。

```
1781            pop     ax                  ; restore the hi-nibble
1782        mov     hinib,al            ;
1783
1784        mov al,#0x2e        ; print a dot
1785        call    display
1786        jmp lfetch          ; try again
```

1786 行：跳转到 lfetch 继续加载 initrd 文件内容。

用流程图表示上述过程如下所示：



3.6.3.3 疑问解答

问题一：从上面的分析可以看出，内核与根文件（initrd）在/boot/map 中的SECTOR_ADDR 数据的磁盘扇区地址都是通过 1053 行的代码：

1053 mov bx,#DESCR0 ; boot the first image

//1053 行：bx 中保存变量 Descr 的地址。

中对 bx 赋值后，通过 bx 获得的。

DESCR0 的定义在 second.S 中如下：

```
4178 #if 0
4179 DESCR0 = DESCR+2
4180 #else
4181 DESCR0 = DESCR
4182 #endif


27 #define DESCR Descr
```

```
    4128 max_secondary:
4129
4130 #if 0
4131 /* the older version 21 layout */
4132 Map        =     max_secondary + SECTOR_SIZE
4133 Dflcmd     =     Map + SECTOR_SIZE
4134 Map2       =     Dflcmd
4135 Descr      =     Dflcmd + SECTOR_SIZE
```

现在的问题是：Descr 是在何时被赋值的？

要解决该问题，要从/sbin/lilo 入手，先看一下相关的数据结构：

```
63 ;*/typedef struct {              /*
 64                                      block     0
 65 ;*/      char name[MAX_IMAGE_NAME+1]; /* image name, NUL
terminated
 66                                 id_name:      .blkb
MAX_IMAGE_NAME_asm+1
 67 ;*/      unsigned short
password_crc[MAX_PW_CRC*(sizeof(INT4)/sizeof(short))];    /* 4 password
CRC-32 values
 68                                 id_password_crc:.blkb
MAX_PW_CRC_asm*4
 69 ;*/      unsigned short rd_size[2]; /* RAM disk size in sectors, 0 if none
 70                                 id_rd_size: .blkb     4          ;don't
change the order !!!
 71 ;*/      SECTOR_ADDR initrd,start;    /* start of initrd & kernel
 72                                 id_initrd:    .blkb     sa_size       ;    **
 73                                 id_start:     .blkb     sa_size       ;    **
 74 ;*/      unsigned short flags,vga_mode; /* image flags & video mode
 75                                 id_flags:     .blkb     2          ;    **
```

```
76                                  id_vga_mode:    .blkb   2           ;   **
77 ;*/} IMAGE_DESCR;                 /*
78                                  id_size:
79                                      endb
```

IMAGE_DESCR 表示内核镜像的一个数据结构，其中 initrd 用来保存 initrd 在/boot/map 上的 SECTOR_ADDR 数据的磁盘扇区地址，id_initrd 是 initrd 在 IMAGE_DESCR 中的偏移量，start 用来保存内核镜像在/boot/map 上的 SECTOR_ADDR 数据的磁盘扇区地址，id_start 是 start 在 IMAGE_DESCR 中的偏移量。

```
164 ;*/typedef struct { /* second stage parameters
165                                     block   0
166 ;*/     char jump[6]; /* jump over the data
167                                 par2_jump:  .blkb   6
168 ;*/     char signature[4]; /* "LILO"
169                                 par2_signature: .blkb   4
170 ;*/     unsigned short version;  /*
171                                 par2_version:   .blkb   2
172 ;*/     unsigned int map_stamp;         /* time of creation of the map
file
173                                 par2_mapstamp:  .blkb   4
174 ;*/     unsigned short stage;    /*
175                                 par2_stage: .blkb   2
176 ;*/     unsigned char port; /* COM port. 0 = none, 1 = COM1, etc. !!!
keep these two serial bytes together !!!
177 ;*/     unsigned char ser_param; /* RS-232 parameters, must be 0 if
unused
178                                 par2_port:  .blkb   1   ; referenced
together
179                                 par2_ser_param: .blkb   1   ; **
```

```
180 ;*/     unsigned short timeout; /* 54 msec delay until input time-out, 0xffff: never
181                                 par2_timeout:    .blkb    2
182 ;*/     unsigned short delay; /* delay: wait that many 54 msec units.
183                                 par2_delay: .blkb    2
184 ;*/     unsigned short msg_len; /* 0 if none
185                                 par2_msg_len:    .blkb    2
186 ;*/     SECTOR_ADDR keytab; /* keyboard translation table
187                                 par2_keytab:     .blkb    sa_size
188 ;*/     unsigned char flag2; /*   flags specific to the second stage loader
189                                 par2_flag2: .blkb    1
190 ;*/} BOOT_PARAMS_2; /* second stage boot loader
191                                     .align   4
192                                 par2_size:
193                                     endb
```

BOOT_PARAMS_2 表示第二引导阶段的数据结构。

second.S 中 154-156 行定义的变量对应 BOOT_PARAMS_2 中的 keytab 变量。

```
154 kt_cx:   .word    0          ; keyboard translation table
 155 kt_dx:   .word    0
 156 kt_al:   .byte    0
```

```
225 ;*/typedef struct {                    /*
226                                 block    0
227 ;*/ char menu_sig[4];     /* "MENU" or "BMP4" signature, or NULs if not present
228                                 mt_sig:      .blkb    4
229 ;*/ unsigned char at_text;    /* attribute for normal menu text
230                                 mt_at_text: .blkb    1
231 ;*/ unsigned char at_highlight; /* attribute for highlighted text
```

```
      232                                 mt_at_hilite:     .blkb     1
      233 ;*/ unsigned char at_border;       /* attribute for borders
      234                                 mt_at_border:     .blkb     1
      235 ;*/ unsigned char at_title;        /* attribute for title
      236                                 mt_at_title:      .blkb     1
      237 ;*/ unsigned char len_title;       /* length of the title string
      238                                 mt_len_title:     .blkb     1
      239 ;*/ char title[MAX_MENU_TITLE+2];     /* MENU title to override
default
      240                                 mt_title:     .blkb
MAX_MENU_TITLE_asm+2
      241 ;*/ short row, col, ncol;          /* BMP row, col, and ncols
      242                                 mt_row:        .blkw     1
      243                                 mt_col:        .blkw     1
      244                                 mt_ncol:       .blkw     1
      245 ;*/ short maxcol, xpitch;          /* BMP max per col, xpitch between
cols
      246                                 mt_maxcol:    .blkw     1
      247                                 mt_xpitch:    .blkw     1
      248 ;*/ short fg, bg, sh;           /* BMP normal text fore, backgr, shadow
      249                                 mt_fg:         .blkw     1
      250                                 mt_bg:         .blkw     1
      251                                 mt_sh:         .blkw     1
      252 ;*/ short h_fg, h_bg, h_sh;        /* highlight fg, bg, & shadow
      253                                 mt_h_fg:       .blkw     1
      254                                 mt_h_bg:       .blkw     1
      255                                 mt_h_sh:       .blkw     1
      256 ;*/ short t_fg, t_bg, t_sh;       /* timer fg, bg, & shadow colors
      257                                 mt_t_fg:       .blkw     1
      258                                 mt_t_bg:       .blkw     1
      259                                 mt_t_sh:       .blkw     1
      260 ;*/ short t_row, t_col;        /* timer position
```

```
261                                     mt_t_row:     .blkw     1
262                                     mt_t_col:     .blkw     1
263 ;*/ short mincol, reserved[3];    /* BMP min per col before spill to next,
reserved spacer
264                                     mt_mincol:    .blkw     1
265                                                   .blkw     3
266
267 ;*/ unsigned int serial_no[MAX_BIOS_DEVICES];     /* Known device
serial nos. 0x80 .. 0x8F
268                                     mt_serial_no:     .blkw
MAX_BIOS_DEVICES_asm*2
269 ;*/ unsigned int raid_offset[MAX_RAID_DEVICES]; /* RAID offsets for
flagged devices
270                                     mt_raid_offset: .blkw
MAX_RAID_DEVICES_asm*2
271 ;*/ unsigned short raid_dev_mask;                  /* 16 bit raid device
mask flagging items in serial_no
272                                     mt_raid_dev_mask: .blkw 1
273 ;*/ SECTOR_ADDR msg; /* initial greeting message
274                                     mt_msg: .blkb     sa_size
275 ;*/ SECTOR_ADDR dflcmd; /* default command line
276                                     mt_dflcmd:    .blkb     sa_size
277 ;*/ SECTOR_ADDR mt_descr[MAX_DESCR_SECS];     /* descriptor
disk addresses
278                                     mt_descr:     .blkb
sa_size*MAX_DESCR_SECS_asm
279 ;*/ char
unused[150-MAX_BIOS_DEVICES*sizeof(int)-(MAX_RAID_DEVICES)*sizeof
(int)-MAX_DESCR_SECS*sizeof(SECTOR_A       DDR)];              /* spacer
280                                     mt_unused:    .blkb
150-sa_size*MAX_DESCR_SECS_asm-4*MAX_BIOS_DEVICES_asm-4*MA
X_RAID_D       EVICES_asm
281 ;*/ short checksum[2];          /* checksum longword
```

```
282                                     mt_cksum:      .blkw     2
283 ;*/ unsigned char mt_flag;          /* contains the FLAG_NOBD only
284                                     mt_flag:       .blkb     1
285 ;*/ char unused2;                   /* spacer beyond checksum
286                                     mt_unused2: .blkb     1
287 ;*/} MENUTABLE;         /* MENU and BITMAP parameters at
KEYTABLE+256
288                                     mt_size:
289                                          endb
MENUTABLE  中 mt_descr 中保存 descr 的磁盘扇区地址。
lilo.c
556 int main(int argc,char **argv)
968         bsect_open(
 969                cfg_get_strg(cf_options,"boot"),
 970                cfg_get_strg(cf_options,"map") ?
 971                     cfg_get_strg(cf_options,"map") : MAP_FILE,
 972                cfg_get_strg(cf_options,"install"),
 973                cfg_get_strg(cf_options,"delay") ?
 974                     timer_number(cfg_get_strg(cf_options,"delay")) : 0,
 975                cfg_get_strg(cf_options,"timeout") ?
 976                     timer_number(cfg_get_strg(cf_options,"timeout")) : -1,
 977                raid_offset );
 978          if (more) {
 979                cfg_init(cf_top);
 980                if (cfg_parse(cf_top)) cfg_error("Syntax error");
 981          }

bsect.c
60 static BOOT_SECTOR bsect,bsect_orig;
   61 static MENUTABLE menuparams;
   62 static DESCR_SECTORS descrs;
   77 static off_t here2;         /* sector address of second stage loader */
```

64 static unsigned char table[SECTOR_SIZE]; /* keytable & params */

568 void bsect_open(char *boot_dev,char *map_file,char *install,int delay,

569 int timeout, int raid_offset)

635 loader = select_loader();

635 行：加载第二引导阶段

636 if (verbose > 0) {

637 printf("Using %s secondary loader\n",

638 loader==&Bitmap ? "BITMAP" :

639 loader==&Third ? "MENU" :

640 "TEXT");

641 }

642 memcpy(&bsect, First.data, MAX_BOOT_SIZE);

643

644 bsect.par_1.timestamp = timestamp;

645 map_begin_section(); /* no access to the (not yet open) map file

646 required, because this map is built in memory */

647 here2 = map_insert_data (loader->data, loader->size);

647 行：将第二引导阶段写入/boot/map 文件，并将其在/boot/map 中的偏移量保存在 here2 中。

memcpy(&param2,loader->data,sizeof(param2));

648 行：将第二引导阶段的数据拷贝到 param2 中。

cfg_parse->cfg_do_set->do_image->boot_image( )

1545 void do_image(void)

1546 {

1547 IMAGE_DESCR descr;

1548 char *name;

1549

```
1550 /*      memset(&descr, 0, sizeof(descr));       Done in
"bsect_common" */
1551        cfg_init(cf_image);
1552        (void) cfg_parse(cf_image);
1553        if (present("image") && initrd_present()) {
1554        bsect_common(&descr, 1);
1555        descr.flags |= FLAG_KERNEL;
1556        name = cfg_get_strg(cf_top,"image");
1557        if (!cfg_get_strg(cf_image,"range")) boot_image(name,&descr);
          //1557 行：将内核和 initrd 的磁盘地址保存在 descr 中。
1558        else
boot_device(name,cfg_get_strg(cf_image,"range"),&descr);
1559        bsect_done(name,&descr);
          //1559 行：将 descr 保存到 descrs 中。
1560        }
1561        cfg_init(cf_top);
1562 }
1380 void bsect_update(char *backup_file, int force_backup, int pass)
1381 {
1382        BOOT_SECTOR bsect_wr;
1383        int temp;
1384 static int timestamp = 0;
1385
1386        if (pass>=0) {
1387        temp = make_backup(backup_file, force_backup, &bsect_orig,
1388                                        boot_dev_nr, "boot sector");
1389        if (temp && !timestamp) bsect.par_1.timestamp = timestamp =
temp;
1390        }
1391
1392 #ifndef LCF_UNIFY
1393 # error "Bios Translation algorithms require '-DUNIFY' in Makefile"
```

```
1394 #endif
1395        if (pass<1) {     /* BIOS_TT logic */
1396        MENUTABLE *menu = &menuparams;
1397        map_descrs(&descrs, menu->mt_descr,
&menuparams.dflcmd);
     //1397:将 descrs 写入/boot/map 的前几个扇区，并将每个扇区的
SECTOR_ADDR 地址保存到 mt_descr 中。
1398        menuparams.raid_dev_mask =
raid_mask((int*)menuparams.raid_offset);
1399        memcpy(menuparams.serial_no, serial_no, sizeof(serial_no));
1400        memcpy(table+256, &menuparams, sizeof(menuparams));
1401        ((int*)table)[SECTOR_SIZE/sizeof(int)-2] = crc32(table,
SECTOR_SIZE-2*sizeof(int), CRC_POLY1);
1402        map_begin_section();
1403        map_add_sector(table);
     //1398-1403 行：将 menuparams 拷贝到 table 中，并将 table 写入到
/boot/map 文件。
1404 #ifdef LCF_FIRST6
1405        /* still use 5 byte address */
1406        (void) map_write(&param2.keytab,1,0,0);
     //1406 行：将 table 在/boot/map 中的磁盘扇区地址写入到第二应道阶段的
keytab 变量中。
1407 #else
1408        (void) map_write(&param2.keytab,1,0);
1409 #endif
1410        map_close(&param2, here2);
     //1407 行：由于 param2 内容改变，需要更新其中/boot/map 中的内容。
1411        }     /* if (pass<1) ...    */
1412
```

总结一下：

second stage 中 keytab 保存 menutables 的磁盘地址。

根据 keytab 可以获得 menutables 的内容。

根据 menutables 中 mt_descr，可以得到前 MAX_DESC_NUM 个扇区的磁盘地址。

根据 mt_descr 可以得到 descrs 的内容。

descrs 中又包含 descr 的内容。

根据 descr 中 start，initrd 的内容又可以得到内核、initrd 的磁盘扇区地址。

4. GRUB

GRUB 全称为 GRand Unified Bootloader。它是一个功能强大的系统引导程序，它可以引导各种开源的操作系统，以及专有操作系统。GRUB 的设计目的是解决个人电脑启动的复杂性。

GRUB 的重要的特征是其灵活性。GRUB 可以识别文件系统和内核可执行文件格式（这点是 LILO 无法做到的）。所以你可以以你喜欢的方式加载任意的操作系统，无需记录你的内核在磁盘上的物理位置。因此，你可以加载内核，只需指定其文件名和和内核所在的驱动器分区。

GRUB 已经有 GRUB 发展到了 GRUB2，这里只讨论 GRUB。下面所有的内容都是基于 GRUB 0.97。

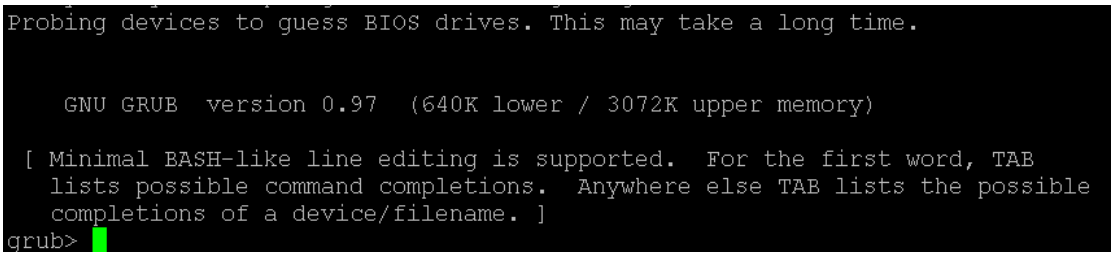
4.1 概述

GRUB 由三部分组成：

stage1：该部分的主要功能是根据字段 stage2_sector 记录的 stage1_5 或者 stage2 的第一个扇区在硬盘上的绝对扇区编号（LBA 模式），加载 stage1_5 或者 stage2 的第一个扇区（该扇区由 start.S 编译后的内容（512 字节）填充）。然后见控制权转交给 start.S

stage1_5：这是一个可选的模块，它可以识别一种文件系统。当 stage1_5 被 stage1 加载到内存获得控制权后，它唯一要做的事就是根据保存 config_file 字段中的驱动器磁盘分区号和 stage2 的文件名，将 stage2 加载到内核中。

stage2：stage2 由 stage1 或者是 stage1_5 加载到内存中，获得控制权后它会根据保存在 config_file 字段中的配置文件的路径去解析配置文件内容，然后加载内核，如果这个过程无误，这直接加载内核。否则会弹出如下图所示的 grub shell，需要用户通过 grub 内置的命令，设定正确的引导参数来启动系统。



和 lilo 一样，这里主要讨论 stage1，stage1_5，stage2 之间是如何交互的，以及通过 grub 内置的 install 命令安装 grub 到系统中的详细过程。

至于 grub 支持文件系统的功能，主要是 stage2 中包含了各种常见文件系统的驱动，所以可以支持文件系统。文件系统驱动本身就是一个较复杂的模块，可以单独讨论，在这里不予分析。

4.2 stage1

4.2.1 源码

stage1.S

片段一：关键变量定义

```
91 stage1_version:
92              .byte   COMPAT_VERSION_MAJOR, COMPAT_VERSION_MINOR
93 boot_drive:
94              .byte   GRUB_INVALID_DRIVE      /* the disk to load stage2 from */
95 force_lba:
96              .byte   0
#stage2_address：stage2 或者 stage1.5 被加载到内存中起始地址
97 stage2_address:
98              .word   0x8000
#stage2_address：stage2 或者 stage1.5 第一个扇区在硬盘上的绝对扇区编号。该指端默认为 1，在 grub 内置的 install 命令会改写该字段。
99 stage2_sector:
100             .long   1
101 stage2_segment:
```

```
102             .word   0x800
```

片段二：加载 **stage2** 或者 **stage1.5** 的 **start.S**

```
130 real_start:
131
132             /* set up %ds and %ss as offset from 0 */
133             xorw    %ax, %ax
134             movw    %ax, %ds
135             movw    %ax, %ss
136
137             /* set up the REAL stack */
138             movw    $STAGE1_STACKSEG, %sp
139
140             sti                     /* we're safe again */
141
142             /*
143              *  Check if we have a forced disk reference here
144              */
145             MOV_MEM_TO_AL(ABS(boot_drive))  /* movb
ABS(boot_drive), %al */
146             cmpb    $GRUB_INVALID_DRIVE, %al
147             je      1f
148             movb    %al, %dl
149 1:
150             /* save drive reference first thing! */
151             pushw   %dx
152
153             /* print a notification message on the screen */
154             MSG(notification_string)
155
156             /* do not probe LBA if the drive is a floppy */
157             testb   $STAGE1_BIOS_HD_FLAG, %dl
```

```
158             jz          chs_mode
159
#检测系统是否支持扩展 BIOS 调用。
160             /* check if LBA is supported */
161             movb        $0x41, %ah
162             movw        $0x55aa, %bx
163             int         $0x13
164
165             /*
166              *  %dl may have been clobbered by INT 13, AH=41H.
167              *  This happens, for example, with AST BIOS 1.04.
168              */
169             popw        %dx
170             pushw       %dx
171
172             /* use CHS if fails */
173             jc          chs_mode
174             cmpw        $0xaa55, %bx
175             jne         chs_mode
176
177             /* check if AH=0x42 is supported if FORCE_LBA is zero */
178             MOV_MEM_TO_AL(ABS(force_lba))     /* movb
ABS(force_lba), %al */
179             testb       %al, %al
#跳转到 lba_mode
180             jnz         lba_mode


184 lba_mode:
185             /* save the total number of sectors */
186             movl        0x10(%si), %ecx
187
```

```
188             /* set %si to the disk address packet */
189             movw    $ABS(disk_address_packet), %si
190
191             /* set the mode to non-zero */
192             movb    $1, -1(%si)
193
194             movl    ABS(stage2_sector), %ebx
195
196             /* the size and the reserved byte */
197             movw    $0x0010, (%si)
198
199             /* the blocks */
200             movw    $1, 2(%si)
201
202             /* the absolute address (low 32 bits) */
203             movl    %ebx, 8(%si)
204
205             /* the segment of buffer address */
206             movw    $STAGE1_BUFFERSEG, 6(%si)
207
208             xorl    %eax, %eax
209             movw    %ax, 4(%si)
210             movl    %eax, 12(%si)
```

186-210：构造 int 13h ah=42h 扩展调用的 dap 参数

```
211
212 /*
213  * BIOS call "INT 0x13 Function 0x42" to read sectors from disk into
memory
214  *      Call with       %ah = 0x42
215  *                      %dl = drive number
216  *                      %ds:%si = segment:offset of disk address
packet
```

```
217  *          Return:
218  *                                  %al = 0x0 on success; err code on failure
219  */
220
221             movb      $0x42, %ah
222             int       $0x13
```

#调用 int 13h ah=42h 的扩展 bios 功能，读取扇区内容

```
223
224             /* LBA read is not supported, so fallback to CHS.   */
225             jc          chs_mode
```

#读取失败，跳到 chs_mode 继续处理

```
226
227             movw      $STAGE1_BUFFERSEG, %bx
```

227 行：为 copy_buffer 做准备，bx 保存源的段地址

```
228             jmp       copy_buffer
```

片段三：拷贝读入的扇区到指定的内存块，跳转到该内存块起始地址。

```
349 copy_buffer:
```

#设置接收区的段地址

```
350             movw      ABS(stage2_segment), %es
351
352             /*
353              * We need to save %cx and %si because the startup code
in
354              * stage2 uses them without initializing them.
355              */
356             pusha
357             pushw     %ds
358
359             movw      $0x100, %cx
```

359 行：拷贝 256 个字，即 512 字节，一个扇区

```
360             movw      %bx, %ds
361             xorw      %si, %si
```

```
362             xorw    %di, %di
363
364             cld
365
366             rep
367             movsw
368
369             popw    %ds
370             popa
371
372             /* boot stage2 */
373             jmp     *(stage2_address)
```

373 行：跳转到 start.S 开始执行

4.2.2 小结

从上面对 stage1.S 源码的分析，可以得出如下结论：

stage1 通过保存在 stage2_sector 中的绝对扇区号，读取该扇区的内容（start.S），将其拷贝到合适的内存，然后跳转到那执行，将控制权交给 start.S

4.3 stage1.5 stage2

在通过 grub 内置的 install 命令安装 grub 时，如果指定使用 stage1.5，stage1 中的 stage2_sector 包保存的是 stage1.5 第一个扇区在磁盘上的绝对扇区编号；如果忽略 stage1.5，stage1 中的 stage2_sector 包保存的是 stage2 第一个扇区在磁盘上的绝对扇区编号。

在 stage2 的 Makefile 文件中，可以看到如下的内容。

```
509 # For e2fs_stage1_5 target.
510 e2fs_stage1_5_exec_SOURCES = start.S asm.S common.c
char_io.c disk_io.c \
511             stage1_5.c fsys_ext2fs.c bios.c
```

生成 stage1.5 的源文件由 start.S asm.S common.c char_io.c disk_io.c stage1_5.c fsys_ext2fs.c bios.c 组成。

其中 start.S 是 stage1.5 的第一个扇区，asm.S 是 staget1.5 的第二个扇区。

```
489 pre_stage2_exec_SOURCES = asm.S bios.c boot.c builtins.c char_io.c \
490             cmdline.c common.c console.c disk_io.c fsys_ext2fs.c \
491             fsys_fat.c fsys_ffs.c fsys_iso9660.c fsys_jfs.c fsys_minix.c \
492             fsys_reiserfs.c fsys_ufs2.c fsys_vstafs.c fsys_xfs.c gunzip.c \
493             hercules.c md5.c serial.c smp-imps.c stage2.c terminfo.c tparm.c
```

```
3216 stage2: pre_stage2 start
3217            -rm -f stage2
3218            cat start pre_stage2 > stage2
```

stage2 的第一扇区和第二扇区也是由 start.S 和 asm.S 组成。

4.3.1 start.S

4.3.1.1 数据结构

```
376 lastlist:
377
378 /*
379  *  This area is an empty space between the main body of code below which
380  *  grows up (fixed after compilation, but between releases it may change
381  *  in size easily), and the lists of sectors to read, which grows down
382  *  from a fixed top location.
383  */
384
```

```
385             .word 0
386             .word 0
387
#BOOTSEC_LISTSIZE=8
388             . = _start + 0x200 - BOOTSEC_LISTSIZE
389
390             /* fill the first data listing with the default */
#blocklist_default_start 记录 stage1.5 或者 stage2 第二扇区（或者是在硬盘上与上一扇区不相邻的扇区）在磁盘上的扇区号。默认值为 2
391 blocklist_default_start:
392             .long 2             /* this is the sector start parameter, in logical
393                                     sectors from the start of the disk, sector 0 */
# blocklist_default_len 记录在磁盘上连续的扇区的数目
394 blocklist_default_len:
395                                  /* this is the number of sectors to read */
396 #ifdef STAGE1_5
397             .word 0             /* the command "install" will fill this up */
398 #else
399             .word (STAGE2_SIZE + 511) >> 9
400 #endif
#上述连续扇区要拷贝到内存的起始端地址
401 blocklist_default_seg:
402 #ifdef STAGE1_5
403             .word 0x220
404 #else
405             .word 0x820         /* this is the segment of the starting address
406                                     to load the data into */
407 #endif
408
```

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

上述的代码块定义了一个记录硬盘上连续扇区块的数据结构，用 C 语言表示如下：

```
typedef sector_address
{
    //起始扇区的在硬盘上的绝对扇区号
    long blocklist_default_start;
    //以 blocklist_default_start 开始，在硬盘上连续的扇区的数目。
    unsigned short blocklist_default_len;
    //上述连续扇区需要拷贝到内存中的起始段地址
    unsigned short    blocklist_default_seg
}
```

虽然代码中给出了 sector_address 一个元素的空间，实际上这里定义了一个数组，这个数组和通常意义上的数组有点不同，其第一个元素为 start.S 的最后 8 个字节。第二个元素为 start.S 的倒数第二个 8 字节…

该数组的生成是在安装 grub 时动态生成的。数组所占字节数为为 512 字节减去 start.S 代码区块的大小

4.3.1.2 Start.S源码

片段一：读取连续的扇区

```
51              .globl    start, _start
52 start:
53 _start:


66              pushw     %dx
67
68              /* print a notification message on the screen */
69              pushw     %si
70              MSG(notification_string)
```

```
 71              popw    %si
 72
 73              /* this sets up for the first run through "bootloop" */
```
将数组中第一个 sector_address 元素地址保存到 di 中。
```
 74              movw    $ABS(firstlist - BOOTSEC_LISTSIZE), %di
 75
 76              /* save the sector number of the second sector in %ebp */
 77              movl    (%di), %ebp
 78
 79              /* this is the loop for reading the secondary boot-loader in */
 80 bootloop:
 81
 82              /* check the number of sectors to read */
 83              cmpw    $0, 4(%di)
 84
 85              /* if zero, go to the start function */
```
所有连续块都读完，跳转到 bootit
```
 86              je      bootit
 87
 88 setup_sectors:
 89              /* check if we use LBA or CHS */
 90              cmpb    $0, -1(%si)
 91
 92              /* jump to chs_mode if zero */
 93              je      chs_mode
 94
 95 lba_mode:
 96             /* load logical sector start */
 97             movl    (%di), %ebx
```
97 行：将起始扇区保存到 ebx 中
```
 98
```

```
 99              /* the maximum is limited to 0x7f because of Phoenix EDD */
100             xorl    %eax, %eax
101             movb    $0x7f, %al
102
103             /* how many do we really want to read? */
104             cmpw    %ax, 4(%di)     /* compare against total
number of sectors */
```

98-104 行：比较连续扇区的数目是否超过 127，EDD 最大限制

```
105
106             /* which is greater? */
107             jg      1f
108
109             /* if less than, set to total */
110             movw    4(%di), %ax
111
112 1:
113             /* subtract from total */
114             subw    %ax, 4(%di)
```

114 行：总数减去 127

```
115
116             /* add into logical sector start */
117             addl    %eax, (%di)
```

117 行：起始扇区号加上 127

```
118
119             /* set up disk address packet */
120
121             /* the size and the reserved byte */
122             movw    $0x0010, (%si)
123
124             /* the number of sectors */
125             movw    %ax, 2(%si)
```

```
126
127             /* the absolute address (low 32 bits) */
128             movl    %ebx, 8(%si)
129
130             /* the segment of buffer address */
131             movw    $BUFFERSEG, 6(%si)
132
133             /* save %ax from destruction! */
134             pushw   %ax
135
136             /* zero %eax */
137             xorl    %eax, %eax
138
139             /* the offset of buffer address */
140             movw    %ax, 4(%si)
141
142             /* the absolute address (high 32 bits) */
143             movl    %eax, 12(%si)
```

119-143 行：设置 dap 参数

```
144
145
146 /*
147  * BIOS call "INT 0x13 Function 0x42" to read sectors from disk into memory
148  *      Call with       %ah = 0x42
149  *                      %dl = drive number
150  *                      %ds:%si = segment:offset of disk address packet
151  *      Return:
152  *                      %al = 0x0 on success; err code on failure
153  */
154
```

```
155             movb    $0x42, %ah
156             int     $0x13
157
158             jc      read_error
159
160             movw    $BUFFERSEG, %bx
161             jmp     copy_buffer
```

片段二：拷贝

```
261 copy_buffer:
262
……
300             cmpw    $0, 4(%di)
301             jne     setup_sectors
```

300-301：连续扇区是否读完，没有则跳转到 setup_sectors 继续读取。

```
302
303             /* update position to load from */
304             subw    $BOOTSEC_LISTSIZE, %di
```

304 行：指向下一个数组中下一个 sector_addr 元素

```
305
306             /* jump to bootloop */
307             jmp     bootloop
```

片段三：启动

```
311 bootit:
312             /* print a newline */
313             MSG(notification_done)
314             popw    %dx     /* this makes sure %dl is our "boot"
drive */
315 #ifdef STAGE1_5
316             ljmp    $0, $0x2200
317 #else /* ! STAGE1_5 */
318             ljmp    $0, $0x8200
```

```
319 #endif /* ! STAGE1_5 */
```

311-319 行：根据是 stage1.5 或者是 stage2，跳转到不同的地方执行。

4.3.1.3 小结

start.S 根据 blocklist 中记录的 stage1.5 或者 stage2 在硬盘上的连续的扇区块，将它们读入内存，然后将控制权交给 stage1.5 或者 stage2。

4.3.2 asm.S

asm.S 是 stage1.5 和 stage2 公有的部分，编译时根据不同的类型给变量设置不同的值。

```
 88 VARIABLE(install_partition)
 89             .long     0xFFFFFF
 90 /* This variable is here only because of a historical reason.    */
 91 VARIABLE(saved_entryno)
 92             .long     0
 93 VARIABLE(stage2_id)
 94             .byte     STAGE2_ID
 95 VARIABLE(force_lba)
 96             .byte     0
 97 VARIABLE(version_string)
 98             .string VERSION
 99 VARIABLE(config_file)
100 #ifndef STAGE1_5
```

如果没定义 1.5，这 config_file 为 menu.lst（该值在通过 install 安装 grub 时可被改写。

```
101             .string "/boot/grub/menu.lst"
102 #else     /* STAGE1_5 */
```

如果是 1.5，需要定义个表示驱动分区号的表示，因为 stage1.5 只能识别一个文件系统，必须给定分区号

```
103             .long     0xffffffff
```

```
104                 .string "/boot/grub/stage2"
105 #endif    /* STAGE1_5 */
```

4.4 GRUB的安装

在前面的讨论中，静态的分析了 stage1.S，start.S，asm.S 的内容，其中多处提到在安装 GRUB 时会改写 stage1.S，start.S，asm.S 生成的二进制内容中的某些字段。现在讨论这个过程的具体内容。

4.4.1 install详解

下面的这段解释摘至 grub.info

```
2744 13.3.18 install
2745 ---------------
2746
2747    -- Command: install [`--force-lba'] [`--stage2=os_stage2_file']
2748                  stage1_file [`d'] dest_dev stage2_file [addr] [`p']
2749                  [config_file] [real_config_file]
2750          This command is fairly complex, and you should not use this command
2751          unless you are familiar with GRUB. Use `setup' (*note setup::)
2752          instead.
2753
2754          In short, it will perform a full install presuming the Stage 2 or
2755          Stage 1.5(1) (*note install-Footnote-1::) is in its final install
2756          location.
2757
2758          In slightly more detail, it will load STAGE1_FILE, validate that
2759          it is a GRUB Stage 1 of the right version number, install in it a
2760          blocklist for loading STAGE2_FILE as a Stage 2. If the option `d'
2761          is present, the Stage 1 will always look for the actual disk
2762          STAGE2_FILE was installed on, rather than using the booting drive.
```

```
2763        The Stage 2 will be loaded at address ADDR, which must be `0x8000'
2764        for a true Stage 2, and `0x2000' for a Stage 1.5. If ADDR is not
2765        present, GRUB will determine the address automatically. It then
2766        writes the completed Stage 1 to the first block of the device
2767        DEST_DEV. If the options `p' or CONFIG_FILE are present, then it
2768        reads the first block of stage2, modifies it with the values of
2769        the partition STAGE2_FILE was found on (for `p') or places the
2770        string CONFIG_FILE into the area telling the stage2 where to look
2771        for a configuration file at boot time. Likewise, if
2772        REAL_CONFIG_FILE is present and STAGE2_FILE is a Stage 1.5, then
2773        the Stage 2 CONFIG_FILE is patched with the configuration file
2774        name REAL_CONFIG_FILE. This command preserves the DOS BPB
(and for
2775        hard disks, the partition table) of the sector the Stage 1 is to
2776        be installed into.
```

4.4.2 install_func剖析

当在 grub>的提示符下输入 install 命令后，grub 最终会调用 install_func 函数来执行该命令。

片段一：变量

```
1742 /* install */
1743 static int
1744 install_func (char *arg, int flags)
1745 {
1746     char *stage1_file, *dest_dev, *file, *addr;
Stage1_buffer：保存 stage1.S 的内容。
1747     char *stage1_buffer = (char *) RAW_ADDR (0x100000);
1748     char *stage2_buffer = stage1_buffer + SECTOR_SIZE;
1749     char *old_sect = stage2_buffer + SECTOR_SIZE;
```

```
    stage2_first_buffer 保存 start.S 的内容
    1750    char *stage2_first_buffer = old_sect + SECTOR_SIZE;
    stage2_second_buffer 保存 asm.S 的内容
    1751    char *stage2_second_buffer = stage2_first_buffer +
SECTOR_SIZE;
    1752    /* XXX: Probably SECTOR_SIZE is reasonable.    */
    1753    char *config_filename = stage2_second_buffer + SECTOR_SIZE;
    1754    char *dummy = config_filename + SECTOR_SIZE;
    1755    int new_drive = GRUB_INVALID_DRIVE;
    1756    int dest_drive, dest_partition, dest_sector;
    1757    int src_drive, src_partition, src_part_start;
    1758    int i;
    1759    struct geometry dest_geom, src_geom;
    1760    int saved_sector;
    stage2_first_sector, stage2_second_sector 保存对应扇区在硬盘上的绝对
扇区号
    1761    int stage2_first_sector, stage2_second_sector;
    1762    char *ptr;
    installaddr 安装地址, installlist blocklist 数组
    1763    int installaddr, installlist;
    1764    /* Point to the location of the name of a configuration file in Stage
2.    */
    1765    char *config_file_location;
    1766    /* If FILE is a Stage 1.5?    */
    1767    int is_stage1_5 = 0;
    1768    /* Must call grub_close?    */
    1769    int is_open = 0;
    1770    /* If LBA is forced?    */
    1771    int is_force_lba = 0;
    1772    /* Was the last sector full? */
    1773    int last_length = SECTOR_SIZE;
```

片段二：读取 **stage1**

```
1880    /* Read Stage 1.  */
1881    is_open = grub_open (stage1_file);
1882    if (! is_open
1883        || ! grub_read (stage1_buffer, SECTOR_SIZE) ==
SECTOR_SIZE)
1884      goto fail;
```

……

片段三：读取 **stage2** 或者 **stage1.5**

```
1932    /* Open Stage 2.  */
1933    is_open = grub_open (file);
1934    if (! is_open)
1935      goto fail;
1936
1937    src_drive = current_drive;
1938    src_partition = current_partition;
1939    src_part_start = part_start;
1940    src_geom = buf_geom;
1941
1942    if (! new_drive)
1943      new_drive = src_drive;
1944    else if (src_drive != dest_drive)
1945      grub_printf ("Warning: the option `d' was not used, but the
Stage 1 will"
1946                   " be installed on a\ndifferent drive than the drive
where"
1947                   " the Stage 2 resides.\n");
1948
1949    /* Set the boot drive.  */
```

```
1950    *((unsigned char *) (stage1_buffer + STAGE1_BOOT_DRIVE)) =
new_drive;
1951
1952    /* Set the "force LBA" flag.  */
1953    *((unsigned char *) (stage1_buffer + STAGE1_FORCE_LBA)) =
is_force_lba;
1954
1955    /* If DEST_DRIVE is a hard disk, enable the workaround, which is
1956       for buggy BIOSes which don't pass boot drive correctly.
Instead,
1957       they pass 0x00 or 0x01 even when booted from 0x80.  */
1958    if (dest_drive & BIOS_FLAG_FIXED_DISK)
1959      /* Replace the jmp (2 bytes) with double nop's.  */
1960      *((unsigned short *) (stage1_buffer +
STAGE1_BOOT_DRIVE_CHECK))
1961        = 0x9090;
```

1950-1961 行：修改 stage1 中的变量。

```
1962
1963    /* Read the first sector of Stage 2.  */
1964    disk_read_hook = disk_read_savesect_func;
1965    if (grub_read (stage2_first_buffer, SECTOR_SIZE) !=
SECTOR_SIZE)
```

1965 行：读取 start.S 的内容

```
1966      goto fail;
1967
1968    stage2_first_sector = saved_sector;
```

1968 行：将 stage2 或者 stage1.5 这硬盘上第一个扇区的绝对扇区号保存到 stage2_first_sector 中。

```
1969
1970    /* Read the second sector of Stage 2.  */
1971    if (grub_read (stage2_second_buffer, SECTOR_SIZE) !=
SECTOR_SIZE)
```

```
1972        goto fail;
1973
1974    stage2_second_sector = saved_sector;
```

1970-1974：读取 stage2 或者 stage1.5 的第二个扇区到 stage2_second_buffer,保存其在硬盘上的绝对扇区号到 stage2_second_sector 中。

……

片段四：修改 stage1 中有关变量

```
1984    /* Check for the Stage 2 id.   */
1985    if (stage2_second_buffer[STAGE2_STAGE2_ID] !=
STAGE2_ID_STAGE2)
1986        is_stage1_5 = 1;
```

1984-1986：判断读入的是否是 stage1.5

```
1987
1988    /* If INSTALLADDR is not specified explicitly in the
command-line,
1989         determine it by the Stage 2 id.   */
1990    if (! installaddr)
1991      {
1992        if (! is_stage1_5)
1993          /* Stage 2.   */
1994          installaddr = 0x8000;
1995        else
1996          /* Stage 1.5.   */
1997          installaddr = 0x2000;
1998      }
1999
```

1990-1998：根据读入的是 stage2 或者是 stage1.5，设置 installaddr 的地址

```
2000    *((unsigned long *) (stage1_buffer +
STAGE1_STAGE2_SECTOR))
2001        = stage2_first_sector;
```

2000-2001：设置 stage1 中 stage2_sector 的值。

```
2002    *((unsigned short *) (stage1_buffer +
STAGE1_STAGE2_ADDRESS))
2003        = installaddr;
2004    *((unsigned short *) (stage1_buffer +
STAGE1_STAGE2_SEGMENT))
2005        = installaddr >> 4;
```

片段五：设置 start.S 中的 blocklist

```
2024    installlist = (int) stage2_first_buffer + SECTOR_SIZE + 4;
```

2024 行：installlist 指向 start.S+512+4 的地方。

```
2025    installaddr += SECTOR_SIZE;
2026
2027    /* Read the whole of Stage2 except for the first sector.    */
2028    grub_seek (SECTOR_SIZE);
```

2028 行：定位到 stage2 或者 stage1.5 的第二个扇区。

```
2029
2030    disk_read_hook = disk_read_blocklist_func;
2031    if (! grub_read (dummy, -1))
2032        goto fail;
```

2029-2032 行：读取 stage2 或者 stage1.5 除去第一个扇区之外的所有内容，并设置 start.S 的 blocklist 内容。

片段六：disk_read_blocklist_func

disk_read_blocklist_func 函数用来设置 blocklist 的内容。

```
1799    /* Write SECTOR to INSTALLLIST, and update INSTALLADDR
and
1800        INSTALLSECT.    */
1801    auto void disk_read_blocklist_func (int sector, int offset, int
length)
1802        {
1803            if (debug)
```

```
1804              printf("[%d]", sector);
1805
1806          if (offset != 0 || last_length != SECTOR_SIZE)
1807            {
1808              /* We found a non-sector-aligned data block. */
1809              errnum = ERR_UNALIGNED;
1810              return;
1811            }
1812
1813          last_length = length;
1814
1815          if (*((unsigned long *) (installlist - 4))
1816              + *((unsigned short *) installlist) != sector/*如果起始块号加上连续块的数目不等于传入的新的块号 sector，说明新的块和之前的块不连续，所以要创建数组的下一个元素，以 sector 为起始块号。*/
1817              || installlist == (int) stage2_first_buffer + SECTOR_SIZE + 4/*第一次进入该函数*/)
1818            {
1819              installlist -= 8;
1819 行：移动到数组中的下一个元素
1820
1821              if (*((unsigned long *) (installlist - 8)))
1821 行：如果下一元素不为 0，表示已经到达了 start.S 的代码段内容，没有空间存储 blocklist，安装 grub 失败（不过这种情况很少发生）。
1822                errnum = ERR_WONT_FIT;
1823              else
1824                {
1825                  *((unsigned short *) (installlist + 2)) = (installaddr >> 4);
1826                  *((unsigned long *) (installlist - 4)) = sector;
1826 行：设置起始块号
1827                }
```

```
1828              }
1829
1830          *((unsigned short *) installlist) += 1;
```
1830 行：如果新的扇区号和之前的连续，这只用将连续扇区号变量加 1。
```
1831          installaddr += 512;
1832        }
```

片段六：设置 **asm.S** 中的 **install_partition** 字段

```
2045    if (*ptr == 'p')
2046      {
2047          *((long *) (stage2_second_buffer +
STAGE2_INSTALLPART))
2048              = src_partition;
```
2047 行：设置 asm.S 中的 install_partition 字段。
```
2049          if (is_stage1_5)
2050            {
2051              /* Reset the device information in FILE if it is a Stage 1.5.
*/
2052              unsigned long device = 0xFFFFFFFF;
2053
```
#config_file_location 是 asm.S 中 config_file 的偏移量
```
2054              grub_memmove (config_file_location, (char *) &device,
2055                            sizeof (device));
```
2053—2055 行：如果是 stage1.5，这重置 asm.S 中的设备信息。
```
2056            }
2057
2058          ptr = skip_to (0, ptr);
2059      }
```

片段七：设置 **asm.S** 中的 **config_file** 中设备信息字段

```
2061    if (*ptr)
```

```
2062        {
2063          grub_strcpy (config_filename, ptr);
2064          nul_terminate (config_filename);
2065
2066          if (! is_stage1_5)
2067            /* If it is a Stage 2, just copy PTR to
CONFIG_FILE_LOCATION.    */
2068            grub_strcpy (config_file_location, ptr);
2069          else
2070            {
2071              char *real_config;
2072              unsigned long device;
2073
2074              /* Translate the external device syntax to the internal
device
2075                 syntax.    */
2076              if (! (real_config = set_device (ptr)))
2077                {
2078                  /* The Stage 2 PTR does not contain the device
name, so
2079                     use the root device instead.    */
2080                  errnum = ERR_NONE;
2081                  current_drive = saved_drive;
2082                  current_partition = saved_partition;
2083                  real_config = ptr;
2084                }
2085
2086              if (current_drive == src_drive)
2087                {
2088                  /* If the drive where the Stage 2 resides is the same
as
```

```
2089                        the one where the Stage 1.5 resides, do not
embed the
2090                        drive number.  */
2091                     current_drive = GRUB_INVALID_DRIVE;
2092                   }
2093
2094                 device = (current_drive << 24) | current_partition;
2095                 grub_memmove (config_file_location, (char *) &device,
2096                               sizeof (device));
```

2096 行：设置 config_file 中的设备信息号。

片段八：将安装过程中修改的内容写回到磁盘中

```
2223       {
2224         /* The first.  */
2225         current_drive = src_drive;
2226         current_partition = src_partition;
2227
2228         if (! open_partition ())
2229           goto fail;
2230
2231         if (! devwrite (stage2_first_sector - src_part_start, 1,
2232                         stage2_first_buffer))
```

2231-2232 行：将 stage2 或者 stage1.5 第一个扇区（start.S）修改了的内容写回到磁盘中。

```
2233           goto fail;
2234
2235         if (! devwrite (stage2_second_sector - src_part_start, 1,
2236                         stage2_second_buffer))
```

2233-2236 行：将 stage2 或者 stage1.5 第二个扇区（asm.S）修改了的内容写回到磁盘中。

```
2237           goto fail;
2238       }
```

```
2239
2240    /* Write the modified sector of Stage 1 to the disk.  */
2241    current_drive = dest_drive;
2242    current_partition = dest_partition;
2243    if (! open_partition ())
2244      goto fail;
2245
2246    devwrite (0, 1, stage1_buffer);
```

2233-2236 行：将 stage1 中修改了的内容写会到 MBR 中。

4.5 实验

下面通过 grub 的 install 安装命令来检验上述分析过程是否正确。

Grub 编译完成后，生成 stage1，e2fs_stage1_5，stage2。

将 stage1，e2fs_stage1_5，stage2 拷贝到/boot/grub 中。

4.5.1 不启用stage1.5

安装时的命令时:install (hd0,0)/boot/grub/stage1 (hd0) (hd0,0)/boot/grub/stage2

在没有安装之前，先查看 stage2 第一个扇区的内容：

```
henry@henry-desktop:/boot/grub$ hexdump -C -n 512 stage2
00000000  52 56 be 03 81 e8 28 01  5e bf f8 81 66 8b 2d 83  |RV....(.^...f.-.|
00000010  7d 04 00 0f 84 ca 00 80  7c ff 00 74 3e 66 8b 1d  |}.......|..t>f..|
00000020  66 31 c0 b0 7f 39 45 04  7f 03 8b 45 04 29 45 04  |f1...9E....E.)E.|
00000030  66 01 05 c7 04 10 00 89  44 02 66 89 5c 08 c7 44  |f.......D.f.\..D|
00000040  06 00 70 50 66 31 c0 89  44 04 66 89 44 0c b4 42  |..pPf1..D.f.D..B|
00000050  cd 13 0f 82 9f 00 bb 00  70 eb 56 66 8b 05 66 31  |........p.Vf..f1|
00000060  d2 66 f7 34 88 54 0a 66  31 d2 66 f7 74 04 88 54  |.f.4.T.f1.f.t..T|
00000070  0b 89 44 0c 3b 44 08 7d  74 8b 04 2a 44 0a 39 45  |..D.;D.}t..*D.9E|
00000080  04 7f 03 8b 45 04 29 45  04 66 01 05 8a 54 0d c0  |....E.)E.f...T..|
00000090  e2 06 8a 4c 0a fe c1 08  d1 8a 6c 0c 5a 52 8a 74  |...L......l.ZR.t|
000000a0  0b 50 bb 00 70 8e c3 31  db b4 02 cd 13 72 46 8c  |.P..p..1.....rF.|
000000b0  c3 8e 45 06 58 c1 e0 05  01 45 06 60 1e c1 e0 04  |..E.X....E.`....|
000000c0  89 c1 31 ff 31 f6 8e db  fc f3 a4 1f be 12 81 e8  |..1.1...........|
000000d0  5e 00 61 83 7d 04 00 0f  85 3c ff 83 ef 08 e9 2e  |^.a.}....<......|
000000e0  ff be 14 81 e8 49 00 5a  ea 00 82 00 00 be 17 81  |.....I.Z........|
000000f0  e8 3d 00 eb 06 be 1c 81  e8 35 00 be 21 81 e8 2f  |.=.......5..!../|
00000100  00 eb fe 4c 6f 61 64 69  6e 67 20 73 74 61 67 65  |...Loading stage|
00000110  32 00 2e 00 0d 0a 00 47  65 6f 6d 00 52 65 61 64  |2......Geom.Read|
00000120  00 20 45 72 72 6f 72 00  bb 01 00 b4 0e cd 10 46  |. Error........F|
00000130  8a 04 3c 00 75 f2 c3 00  00 00 00 00 00 00 00 00  |..<.u...........|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  02 00 00 00 c6 00 20 08  |.............. .|
00000200
```

从图上可以看出，没安装之前，stage2 的 start.S 中。

Blocklist 中只有一个元素：

扇区的起始地址为：0x00000002

连续扇区的长度：0x00c6

内存起始地址为：0x0802

下面通过 grub install 安装 grub，并查看安装过程中的调试信息：

```
henry@henry-desktop:~/grub-0.97/grub$ sudo ./grub --verbose
Probing devices to guess BIOS drives. This may take a long time.
Attempt to open drive 0x80 (/dev/sda)


    GNU GRUB  version 0.97  (640K lower / 3072K upper memory)

 [ Minimal BASH-like line editing is supported.  For the first word, TAB
   lists possible command completions.  Anywhere else TAB lists the possible
   completions of a device/filename. ]
grub> install (hd0,0)/boot/grub/stage1 (hd0) (hd0,0)/boot/grub/stage2
install (hd0,0)/boot/grub/stage1 (hd0) (hd0,0)/boot/grub/stage2
<sector,byte_offset,byte_len><2457744, 0, 512>
disk_read_savesect_func [2459792]
<sector,byte_offset,byte_len><2457745, 0, 512>
disk_read_savesect_func [2459793]
<sector,byte_offset,byte_len><2457745, 0, 3584>
disk_read_blocklist_func [2459793]
the number of sector=1
disk_read_blocklist_func [2459794]
the number of sector=2
disk_read_blocklist_func [2459795]
the number of sector=3
```

图只标红的地方表示 blocklist 中第一个元素：

起始扇区地址为：2459793=0x258891

```
                        disk_read_blocklist_func [2459884]
the number of sector=92
disk_read_blocklist_func [2459885]
the number of sector=93
disk_read_blocklist_func [2459886]
the number of sector=94
disk_read_blocklist_func [2459887]
the number of sector=95
<sector,byte_offset,byte_len><2457840, 0, 4096>
<sector,byte_offset,byte_len><2457848, 0, 4096>
disk_read_blocklist_func [2459896]
the number of sector=1
disk_read_blocklist_func [2459897]
the number of sector=2
disk_read_blocklist_func [2459898]
```

连续扇区的数目为 95=0x5F

第二个元素

起始扇区地址为：2459793=0x2588F8

```
the number of sector=100
disk_read_blocklist_func [2459996]
the number of sector=101
disk_read_blocklist_func [2459997]
the number of sector=102
disk_read_blocklist_func [2459998]
the number of sector=103
Write 1 sectors starting from 0 sector to drive 0x80 (/dev/sda)
eb489010 8ed0bc00 b0b80000 8ed88ec0 fbbe007c bf0006b9 0002f3a4 ea210600
00bebe07 3804750b 83c61081 fefe0775 f3eb16b4 02b001bb 007cb280 8a740302
ff000080 90882500 0008fa90 90f6c280 7502b280 ea597c00 0031c08e d88ed0bc
0020fba0 407c3cff 740288c2 52be7f7d e83401f6 c2807454 b441bbaa 55cd135a
```

连续扇区的数目为 103=0x67

安装后在查看 stage2 的第一个扇区内容：

```
henry@henry-desktop:/boot/grub$ hexdump -C -n 512 stage2
00000000  52 56 be 03 81 e8 28 01  5e bf f8 81 66 8b 2d 83  |RV....(.^...f.-.|
00000010  7d 04 00 0f 84 ca 00 80  7c ff 00 74 3e 66 8b 1d  |}.......|..t>f..|
00000020  66 31 c0 b0 7f 39 45 04  7f 03 8b 45 04 29 45 04  |f1...9E....E.)E.|
00000030  66 01 05 c7 04 10 00 89  44 02 66 89 5c 08 c7 44  |f.......D.f.\..D|
00000040  06 00 70 50 66 31 c0 89  44 04 66 89 44 0c b4 42  |..pPf1..D.f.D..B|
00000050  cd 13 0f 82 9f 00 bb 00  70 eb 56 66 8b 05 66 31  |........p.Vf..f1|
00000060  d2 66 f7 34 88 54 0a 66  31 d2 66 f7 74 04 88 54  |.f.4.T.f1.f.t..T|
00000070  0b 89 44 0c 3b 44 08 7d  74 8b 04 2a 44 0a 39 45  |..D.;D.}t..*D.9E|
00000080  04 7f 03 8b 45 04 29 45  04 66 01 05 8a 54 0d c0  |....E.)E.f...T..|
00000090  e2 06 8a 4c 0a fe c1 08  d1 8a 6c 0c 5a 52 8a 74  |...L......l.ZR.t|
000000a0  0b 50 bb 00 70 8e c3 31  db b4 02 cd 13 72 46 8c  |.P..p..1.....rF.|
000000b0  c3 8e 45 06 58 c1 e0 05  01 45 06 60 1e c1 e0 04  |..E.X....E.`....|
000000c0  89 c1 31 ff 31 f6 8e db  fc f3 a4 1f be 12 81 e8  |..1.1...........|
000000d0  5e 00 61 83 7d 04 00 0f  85 3c ff 83 ef 08 e9 2e  |^.a.}....<......|
000000e0  ff be 14 81 e8 49 00 5a  ea 00 82 00 00 be 17 81  |.....I.Z........|
000000f0  e8 3d 00 eb 06 be 1c 81  e8 35 00 be 21 81 e8 2f  |.=.......5..!../|
00000100  00 eb fe 4c 6f 61 64 69  6e 67 20 73 74 61 67 65  |...Loading stage|
00000110  32 00 2e 00 0d 0a 00 47  65 6f 6d 00 52 65 61 64  |2......Geom.Read|
00000120  00 20 45 72 72 6f 72 00  bb 01 00 b4 0e cd 10 46  |. Error........F|
00000130  8a 04 3c 00 75 f2 c3 00  00 00 00 00 00 00 00 00  |..<.u...........|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  f8 88 25 00 67 00 00 14  91 88 25 00 5f 00 20 08  |..%.g.....%._. .|
00000200
```

可以看到，blocklist 现在由两个元素：

元素一：

起始扇区地址：0x00258891

连续扇区长度：0x005f

元素二：

起始扇区地址：0x002588f8

连续扇区长度：0x0067

可以看出，这和调试中输出的信息吻合。

4.5.2 启用stage1.5

安装时的命名：

install (hd0,0)/boot/grub/stage1 (hd0) (hd0,0)/boot/grub/e2fs_stage1_5 p (hd0,0)/boot/grub/stage2

没安装之前 e2fs_stage1_5 的前 2 个扇区的内容:

```
00000000  52 56 be 03 21 e8 2a 01  5e bf f8 21 66 8b 2d 83  |RV..!.*.^..!f.-.|
00000010  7d 04 00 0f 84 ca 00 80  7c ff 00 74 3e 66 8b 1d  |}.......|..t>f..|
00000020  66 31 c0 b0 7f 39 45 04  7f 03 8b 45 04 29 45 04  |f1...9E....E.)E.|
00000030  66 01 05 c7 04 10 00 89  44 02 66 89 5c 08 c7 44  |f.......D.f.\..D|
00000040  06 00 70 50 66 31 c0 89  44 04 66 89 44 0c b4 42  |..pPf1..D.f.D..B|
00000050  cd 13 0f 82 9f 00 bb 00  70 eb 56 66 8b 05 66 31  |........p.Vf..f1|
00000060  d2 66 f7 34 88 54 0a 66  31 d2 66 f7 74 04 88 54  |.f.4.T.f1.f.t..T|
00000070  0b 89 44 0c 3b 44 08 7d  74 8b 04 2a 44 0a 39 45  |..D.;D.}t..*D.9E|
00000080  04 7f 03 8b 45 04 29 45  04 66 01 05 8a 54 0d c0  |....E.)E.f...T..|
00000090  e2 06 8a 4c 0a fe c1 08  d1 8a 6c 0c 5a 52 8a 74  |...L......l.ZR.t|
000000a0  0b 50 bb 00 70 8e c3 31  db b4 02 cd 13 72 46 8c  |.P..p..1.....rF.|
000000b0  c3 8e 45 06 58 c1 e0 05  01 45 06 60 1e c1 e0 04  |..E.X....E.`....|
000000c0  89 c1 31 ff 31 f6 8e db  fc f3 a4 1f be 14 21 e8  |..1.1.........!.|
000000d0  60 00 61 83 7d 04 00 0f  85 3c ff 83 ef 08 e9 2e  |`.a.}....<......|
000000e0  ff be 16 21 e8 4b 00 5a  ea 00 22 00 00 be 19 21  |...!.K.Z.."....!|
000000f0  e8 3f 00 eb 06 be 1e 21  e8 37 00 be 23 21 e8 31  |.?.....!.7..#!.1|
00000100  00 eb fe 4c 6f 61 64 69  6e 67 20 73 74 61 67 65  |...Loading stage|
00000110  31 2e 35 00 2e 00 0d 0a  00 47 65 6f 6d 00 52 65  |1.5......Geom.Re|
00000120  61 64 00 20 45 72 72 6f  72 00 bb 01 00 b4 0e cd  |ad. Error.......|
00000130  10 46 8a 04 3c 00 75 f2  c3 00 00 00 00 00 00 00  |.F..<.u.........|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  02 00 00 00 00 00 20 02  |.............. .|
00000200  ea 70 22 00 00 00 03 02  ff ff ff 00 00 00 00 00  |.p".............|
00000210  02 00 30 2e 39 37 00 ff  ff ff ff 2f 62 6f 6f 74  |..0.97...../boot|
00000220  2f 67 72 75 62 2f 73 74  61 67 65 32 00 00 00 00  |/grub/stage2....|
00000230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

安装过程中的调试信息如下：

```
henry@henry-desktop:~/grub-0.97/grub$ sudo ./grub
Probing devices to guess BIOS drives. This may take a long time.


    GNU GRUB  version 0.97  (640K lower / 3072K upper memory)

 [ Minimal BASH-like line editing is supported.  For the first word, TAB
   lists possible command completions.  Anywhere else TAB lists the possible
   completions of a device/filename. ]
grub> install (hd0,0)/boot/grub/stage1 (hd0) (hd0,0)/boot/grub/e2fs_stage1_5 p (hd0,0)/boot/grub/stage2
install (hd0,0)/boot/grub/stage1 (hd0) (hd0,0)/boot/grub/e2fs_stage1_5 p (hd0,0)/boot/grub/stage2
<sector,byte_offset,byte_len><2457952, 0, 512>
disk_read_savesect_func [2460000]
<sector,byte_offset,byte_len><2457953, 0, 512>
disk_read_savesect_func [2460001]
<sector,byte_offset,byte_len><2457953, 0, 3584>
disk_read_blocklist_func [2460001]
the number of sector=1
disk_read_blocklist_func [2460002]
the number of sector=2
disk_read_blocklist_func [2460003]
the number of sector=3
disk_read_blocklist_func [2460004]
the number of sector=4
disk_read_blocklist_func [2460005]
```

Blocklist 中第一个元素：

开始扇区：2460001 = 0x258961

```
disk_read_blocklist_func [2460016]
the number of sector=16
<sector,byte_offset,byte_len><2457745, 0, 512>
disk_read_savesect_func [2459793]
Write 1 sectors starting from 0 sector to drive 0x80 (/dev/sda)
eb489010 8ed0bc00 b0b80000 8ed88ec0 fbbe007c bf0006b9 0002f3a4 ea210600
00bebe07 3804750b 83c61081 fefe0775 f3eb16b4 02b001bb 007cb280 8a740302
```

连续扇区的数目：16=0x10

可以看到，blocklist 中只有一个元素。

安装之后 e2fs_stage1_5 的前 2 个扇区的内容:

```
henry@henry-desktop:/boot/grub$ hexdump -C -n 1024 e2fs_stage1_5
00000000  52 56 be 03 21 e8 2a 01  5e bf f8 21 66 8b 2d 83  |RV..!.*.^..!f.-.|
00000010  7d 04 00 0f 84 ca 00 80  7c ff 00 74 3e 66 8b 1d  |}.......|..t>f..|
00000020  66 31 c0 b0 7f 39 45 04  7f 03 8b 45 04 29 45 04  |f1...9E....E.)E.|
00000030  66 01 05 c7 04 10 00 89  44 02 66 89 5c 08 c7 44  |f.......D.f.\..D|
00000040  06 00 70 50 66 31 c0 89  44 04 66 89 44 0c b4 42  |..pPf1..D.f.D..B|
00000050  cd 13 0f 82 9f 00 bb 00  70 eb 56 66 8b 05 66 31  |........p.Vf..f1|
00000060  d2 66 f7 34 88 54 0a 66  31 d2 66 f7 74 04 88 54  |.f.4.T.f1.f.t..T|
00000070  0b 89 44 0c 3b 44 08 7d  74 8b 04 2a 44 0a 39 45  |..D.;D.}t..*D.9E|
00000080  04 7f 03 8b 45 04 29 45  04 66 01 05 8a 54 0d c0  |....E.)E.f...T..|
00000090  e2 06 8a 4c 0a fe c1 08  d1 8a 6c 0c 5a 52 8a 74  |...L......l.ZR.t|
000000a0  0b 50 bb 00 70 8e c3 31  db b4 02 cd 13 72 46 8c  |.P..p..1.....rF.|
000000b0  c3 8e 45 06 58 c1 e0 05  01 45 06 60 1e c1 e0 04  |..E.X....E.`....|
000000c0  89 c1 31 ff 31 f6 8e db  fc f3 a4 1f be 14 21 e8  |..1.1.........!.|
000000d0  60 00 61 83 7d 04 00 0f  85 3c ff 83 ef 08 e9 2e  |`.a.}....<......|
000000e0  ff be 16 21 e8 4b 00 5a  ea 00 22 00 00 be 19 21  |...!.K.Z.."....!|
000000f0  e8 3f 00 eb 06 be 1e 21  e8 37 00 be 23 21 e8 31  |.?.....!.7..#!.1|
00000100  00 eb fe 4c 6f 61 64 69  6e 67 20 73 74 61 67 65  |...Loading stage|
00000110  31 2e 35 00 2e 00 0d 0a  00 47 65 6f 6d 00 52 65  |1.5......Geom.Re|
00000120  61 64 00 20 45 72 72 6f  72 00 bb 01 00 b4 0e cd  |ad. Error.......|
00000130  10 46 8a 04 3c 00 75 f2  c3 00 00 00 00 00 00 00  |.F..<.u.........|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  61 89 25 00 10 00 20 02  |........a.%... .|
00000200  ea 70 22 00 00 00 03 02  ff ff 00 00 00 00 00 00  |.p".............|
00000210  02 00 30 2e 39 37 00 ff  ff 00 ff 2f 62 6f 6f 74  |..0.97...../boot|
00000220  2f 67 72 75 62 2f 73 74  61 67 65 32 00 00 00 00  |/grub/stage2....|
00000230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

可以看出，安装之后，e2fs_stage1_5 的 blocklisk 中只有一个元素

起始扇区：0x00258961

连续扇区的数目：0x0010

且 asm.S 中 config_file 字段中的驱动分区号变为 0xff00ffff

和调试信息吻合。

4.5.3 注意

上述实验过程中，在通过 install 安装 GRUB 后，stage1.5 和 stage2 在硬盘上的内容并不会在安装之后马上修改（正常情况应该是马上修改，这可以算是 grub0.97 的一个 bug 吧。），原因是内存中缓存的数据并没有同步到硬盘上，要看到 stage1.5 和 stage2 的修改，必须重启系统后再查看。

4.6 小结

通过上面对 GRUB 的分析，GRUB 的处理流程如下：

1、stage1(stage1.S)根据安装 grub 时设置的 stage2_sector 字段，读取 stage1.5 或者 stage2 的第一个扇区内容(start.S)。

2、如果安装时没有选用 stage1.5，则 start.S 根据保存在 blocklist 中的数据，从硬盘上指定的扇区读取 stage2 到内存，然后将控制权交给 stage2。

stage2 先查找是否有配置文件 menu.lst（或者安装时指定的配置文件），如果没有或者解析配置文件失败，则显示 grub 提示界面，需要用户根据 grub 内置命令启动系统。否则直接加载系统。

3、如果安装时选用 stage1.5，则 start.S 根据保存在 blocklist 中的数据，从硬盘上指定的扇区读取 stage1.5 到内存，stage1.5 根据 config_file 中配置，读取 stage2 到内存，然后将控制权交给 stage2。

5. 参考资料

[1] linux 内核 0.11 源码
[2] lilo-22.8 源码
[3] grub 0.97 源码
[4] BIOS 中断大全（网络资源）
[5] 扩展 int 13 调用（网络资源）
[6] linux 内核 0.11 详细注释 赵炯
[7] grub 源代码分析 （网络资源）
[8] 深入了解 linux 内核 （第三版）
[9] linxu 内核源代码情景分析 毛德超
[10] 汇编语言程序设计 Richard Blum 著 马朝晖 等译
[11] IBM PC 汇编语言程序设计 （第五版） Peter Abel 著 沈美明 温冬婵 译
[12] Orange’s 一个操作系统的实现 于渊