GRUB 源代码分析

资料来源：

http://linuxsir.org/bbs/thread170348.html

合集者：tbfly （tbfly@smth， luzhihe@88）

下文均是上面链接各网友精彩发帖的集合，除了明显的错误修改外，并无内容修改。若有出入请参考原链接。

- **phyma**:

基于 0.93 grub。

其实总体上我们可以把 grub 看成一个 mini os，他有 shell，支持 script，有文件系统……我们可以把 stage1, stage1.5 看成一个 boot loader，而 stage2 则是一个 os，只不过这个 os 是专门 load 其他 os 的 os，为此，stage2 支持像 kernel，initrd，chainloader 等等为此目的而设置的内部“命令”……

1.简略流程

（如果你不熟悉 grub，建议您先看完这这一节再向后）

a，Bios 执行 int 0x19，加载 MBR 至 0x7c00 并跳转执行。

如果你安装 grub 到 mbr，grub 的安装程序会把 stage1（512B）拷贝到 mbr。

视 stage2 的大小，安装程序会在 stage1 中嵌入 stage1.5 或者 stage2 的磁盘位置信息。

b，stage1 开始执行，它在进行直接加载 stage1.5 或者 stage2 并跳转执行（还有些额外的工作，这里忽略先）

stage1.5 很无辜，除了加载 stage2 以外它没有其他任何作用。

所以，不论是哪种情况，这一步的结果都是 stage2 开始运行了。

c，stage2 这个 mini os 终于开始正式运行了！它会把系统切入保护模式，设置好 C 运行环境（主要是 bss）

他会找 config 文件先（就是我们的 menulist），如果没有的话就执行一个 shell，等待我们输入命令。

然后 grub 的工作就是输入命令-解析命令-执行命令的循环，当然 stage2 本身也很无辜，他是为加载其他 os 而存在的，所以如果情况允许，在他执行 boot 命令以后就会把控制权转交出去！

2.stage2-文件系统

a，调用接口

类似系统调用，在 stage2/disk_io.c 中定义了 grub_open,grub_close,grub_read,grub_dir

全局函数用于 stage2 的文件操作:打开，关闭，读，切换目录。

为了简化文件系统驱动的编写，grub 不支持磁盘写（对于一个 loader 来说也没有必要去写磁盘）

b，文件系统驱动接口

任何一个文件系统驱动必须在 fsys_table（stage2/disk_io.c）数组中去放置一个

struct fsys_entry 的结构体，定义如下（stage2/filesys.h）：

```
struct fsys_entry
{
  char *name;                          //文件系统名称
  int (*mount_func) (void);       //初始化函数
  int (*read_func) (char *buf, int len);     //文件读函数
  int (*dir_func) (char *dirname);          //目录/文件打开函数
  void (*close_func) (void);                 //文件关闭函数
  int (*embed_func) (int *start_sector, int needed_sectors);   //? 大部分驱动设置此项为 NULL
};
```

c，情景：读入文件

假设在 grub 的 menulist 中有：kernel (hd0,0)/boot/vmlinuz（或者我们在 grub shell 中执行此命令）

stage2 会先调用 grub_open("(hd0,0)/boot/vmlinuz") 来打开文件。

在执行这个函数中，grub 会先在 fsys_table 中循环调用 fsys_entry::mount_func 去发现一个返回值为真的文件系统，即为当前的文件系统。然后利用当前文件系统驱动的 fsys_entry::dir_func 去打开/boot/vmlinuz

然后 stage2 会调用 grub_read(buf,0)读入全部的/boot/vmlinuz 文件至内存中的 buf 地址

grub_read 是 fsys_entry::read_func 的封装。

最后 stage2 回调用 grub_close 关闭文件，跟前面一样，这个调用仅仅是当前文件系统 fsys_entry::close_func 的简单封装。

d，文件系统驱动

你可以在 stage2/fsys_*.c 找到各个文件系统驱动，当然你如果对某个文件的结构熟悉，而 grub 中又没有相应的驱动，你可以按照 b 节中方法向 grub 中添加此文件系统的支持，比如 NTFS，你只要实现 mount（判断是否是你所支持的文件系统），dir（打开文件或者目录），read（读），close（关闭文件）功能就好。

3. stage2-Shell

大部分情况我们是通过一个 menulist 文件来控制 grub，这样 grub 会分析此文件的内容，然后显示一个菜单让我们选择。但是我不打算介绍 grub stage2 时怎样显示这个菜单的，这不是 grub 特殊的地方，这一节将对菜单选择执行的底层机制做小小的分析。（在 grub 没有找到 menulist 的时候，它就会执行一个 shell，你完全可以通过这里内置的命令来控制 grub，实际上你可以把 menulist 当作一个脚本文件，它也完全是通过内置命令执行的）

a，stage2 流程

stage1 或者 stage1.5 加载完 stage2 后，会跳转至 stage2 执行，stage2 的入口是 stage2/asm.S asm.S 在设置好 C 运行环境之后，会调用第一个 C 函数 init_bios_info（stage2/common.c），这个函数在执行一些底层的初始化之后，会调用 stage2 的 main 函数 cmain（stage2/stage2.c），这样 stage2 这个 mini os 正式开始运行了！

针对 menu.lst 和 shell 这两种情况，cmain 将：

menu.lst: run_menu()(stage2.c)->run_script()(cmdline.c)->find_command->执行命令函数

shell: enter_cmdline()(cmdline.c)->find_command->执行命令函数

殊途同归，最后都归结为命令行的解释执行 find_command（stage2/cmdline.c）按照 menu.lst 中或者 shell 用户输入的命令字符串，在一个全局性 struct builtin *builtin_table[](stage2/builtin.c)变量中去找到内置命令的函数，然后执行。

值得一提的是 grub 的 shell 类似 bash 的命令补全和命令历史纪录。

b，内部数据结构

```
struct builtin
{
  /* 命令名称，重要，是搜索命令时的依据 */
  char *name;
  /* 命令函数，重要，是搜索匹配后调用的函数 */
  int (*func) (char *, int);
  /* 功能标示，一般未用到. */
  int flags;
  /* 简短帮助信息 */
  char *short_doc;
  /* 完整帮助信息 */
  char *long_doc;
};
```

struct builtin *builtin_table[]；(stage2/builtin.c)

c，Hack 提示

编写一个自己的 grub 命令。只需按照 b 节所述，填充一个 struct builtin 结构，按后将其指针放入 builtin_table 指针数组，你就可以在 menu.lst 或者 grub shell 中使用这个命令了。

4.TODO

- **home_king:**

引用:

> 作者: phyma
>
> 视 stage2的大小，安装程序会在 stage1 中嵌入 stage1.5或者 stage2的磁盘位置信息。
> b，stage1开始执行，它在进行直接加载 stage1.5或者 stage2并跳转执行（还有些额外的工作，这里忽略先）
> stage1.5很无辜，除了加载 stage2以外它没有其他任何作用。

针对上面，我来修正一下。

stage1的代码文件，是源码目录下 stage1/stage1.S，汇编后便成了一个512字节的 img，被写在硬盘的**0面0道第1扇区**，作为硬盘的主引导扇区。

注意，硬盘主引导扇区 = 硬盘主引导记录（MBR）+ 硬盘分区表（DPT）

stage1的工作并不是加载什么 stage1.5或者 stage2，而是加载**0面0道第2扇区**上的512字节代码至0x8000，然后跳至0x8000执行。这里我们提及的另一个512字节代码，是来自源码目录下 stage2/start.S 文件的，而 start.S 的作用是作为 stage1.5或者 stage2(视乎编译 grub 时的指定)的总入口，它才是 stage1.5或者 stage2的真正加载器。

总结起来，那么就是 stage1加载 start，然后将执行权交给 start，由start 来加载 stage1.5或者 stage2。所以，斑竹说的“安装程序会在 stage1 中嵌入 stage1.5或者 stage2的磁盘位置信息”这句话是错误的。

大家想验证这些关系，请使用以下命令(这里假设你把 grub 安装在第一个 IDE 硬盘):

代码:

1. 倾印 stage1.S 的机器代码映像 dd if=/dev/hda of=BOOT.img bs=1 count=512
2. 倾引 start.S 的机器代码映像 dd if=/dev/hda of=START.img skip=512 bs=1 count=512
3. 用 emacs 打开这些映像，并将它们转换为16进制格式来查看 emacs xxx.img M-x hexlify-buffer 同时对照一下源码中的内容，尤其是那些"GRUB"、"Loading stage1.5"等静态显示数据，你就会验证了上面我的说法，并且在 start.S 中找到更多的内幕。

stage2的内幕并不神奇，正如斑竹所言，是一个 mini OS，但我们觉得 GRUB 神奇的地方在于文件系统的识别功能。这种神奇色彩也就是"鸡蛋与鸡谁先有"的矛盾体。

PC 上电后，就会执行 BIOS 的代码，BIOS 将加载硬盘主引导扇区，总共512字节的二进制代码，这些代码就是 stage1，然后 BIOS 将执行 stage1。这一点大家都不会感到奇怪，这是很自然的流程。然而，stage2的体积比较大(因为它实现的功能比较全面嘛)，所以一般 stage2不会被放在固定的磁盘扇区中以供 stage1只使用 BIOS 例程便可对其 raw read，那么，stage2就会作为一个文件被放在文件系统里。

大家都知道，stage1.5就是文件系统的支撑代码，在 stage1.5没有被加载以前，stage2是不能被 stage1找到的，所以我们研究 GRUB，关键看看究竟 stage1是怎么加载 stage1.5的，而 stage1.5又被放在哪里。

接着上面的话题。
stage1.5 究竟被放在哪呢？很多兄弟可能以为它就是/boot/grub/底下的哪些 xxfs_stage1_5 文件，但试想一下，要找到 boot 分区所在的 stage1_5 文件，那么就必须使得 stage1 具备文件系统识别功能，而stage1.5 本身就是文件系统的支撑代码，它必须加载 stage1.5 才能具备这种功能。那么，我们又回到了那种矛盾体的悖论——要加载 stage1.5 来找到 stage1.5？ 呵呵。

所以用来识别 boot 分区文件系统的 stage1_5 不能作为文件来被 stage1 读取，它只能被存放在固定的扇区中。这里强调"用来识别 boot 分区文件系统"，那是因为并不是所有的 stage1.5文件都被放在固定扇区的，只有 boot 分区的文件系统对应的 stage1.5 才会被放在固定的扇

区中去！比如说，你的 boot 分区的文件系统是 ext2，那么在安装 GRUB 的 stage1 的时候，e2fs_stage1_5 就会被存放至一个固定的扇区集，而其他的如 reiserfs_stage1_5 就依然作为文件来存放，以供 GRUB 使用 root()命令来识别其他的 boot 分区(那时候，stage2 已经被加载了，所以这个不成问题)。

那么，如何验证我上面的说法呢？还是使用 dd 命令。

代码:

1. dd if=/dev/hda of=STAGE1_5.img bs=1k skip=1 count=20 将 STAGE1_5.img 用 emacs 打开，转换为 hex 格式查看。
2. 将/boot/grub/e2fs_stage1_5拷贝一份到当前目录，用 emacs 打开，转换为 hex 格式查看。注意，如果你的 boot 分区是别的文件系统，应该打开对应的 stage1_5文件来查看。我这里的 boot 分区为 ext2文件系统。

查找这两个文件中相同的字符串，如"Loading stage1.5"，"GRUB" 等，同时注意到它们交集的行数数量，你会发现，原来 e2fs_stage1_5被放在0面0道的第3个扇区开始往后10多 K 的扇区集里。

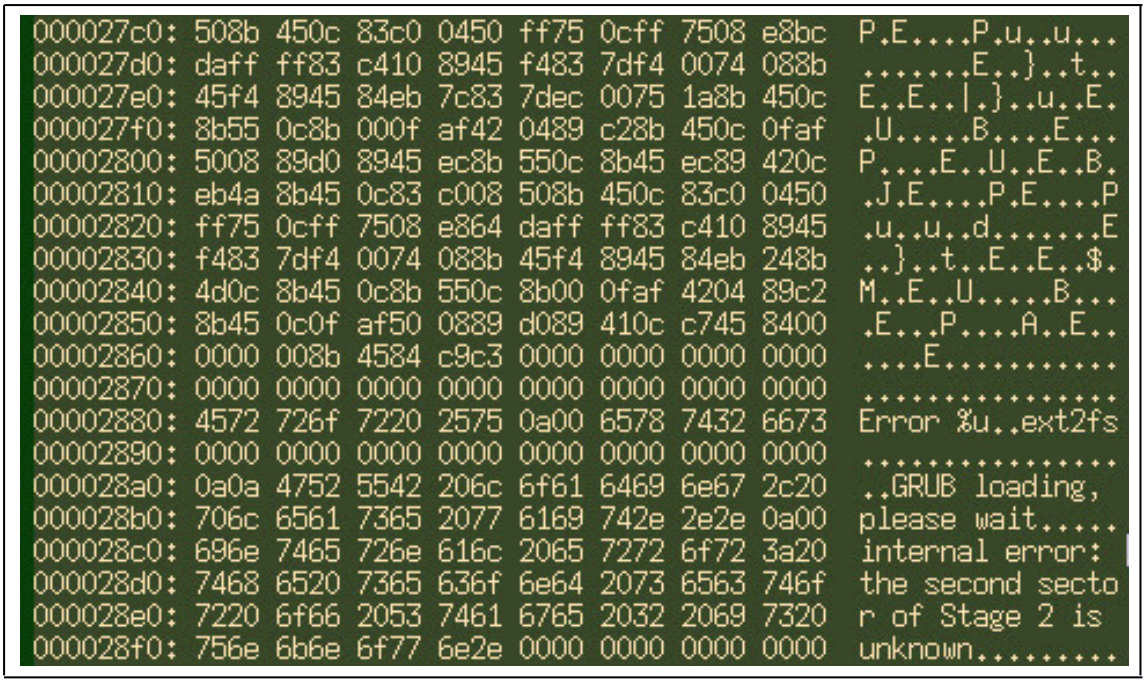
这里是 e2fs_stage1_5文件的 hex 倾印片断：

```
00000200: ea70 2200 0000 0302 ffff ff00 0000 0000  .p".............
00000210: 0200 302e 3934 00ff ffff ff2f 626f 6f74  ..0.94...../boot
00000220: 2f67 7275 622f 7374 6167 6532 0000 0000  /grub/stage2....
00000230: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
00002a00: 5008 89d0 8945 ec8b 550c 8b45 ec89 420c  P....E..U..E..B.
00002a10: eb4a 8b45 0c83 c008 508b 450c 83c0 0450  .J.E....P.E....P
00002a20: ff75 0cff 7508 e864 daff ff83 c410 8945  .u..u..d.......E
00002a30: f483 7df4 0074 088b 45f4 8945 84eb 248b  ..}..t..E..E..$.
00002a40: 4d0c 8b45 0c8b 550c 8b00 0faf 4204 89c2  M..E..U.....B...
00002a50: 8b45 0c0f af50 0889 d089 410c c745 8400  .E...P....A..E..
00002a60: 0000 008b 4584 c9c3 0000 0000 0000 0000  ....E...........
00002a70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002a80: 4572 726f 7220 2575 0a00 6578 7432 6673  Error %u..ext2fs
00002a90: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002aa0: 0a0a 4752 5542 206c 6f61 6469 6e67 2c20  ..GRUB loading,
00002ab0: 706c 6561 7365 2077 6169 742e 2e2e 0a00  please wait.....
00002ac0: 696e 7465 726e 616c 2065 7272 6f72 3a20  internal error:
00002ad0: 7468 6520 7365 636f 6e64 2073 6563 746f  the second secto
00002ae0: 7220 6f66 2053 7461 6765 2032 2069 7320  r of Stage 2 is
00002af0: 756e 6b6e 6f77 6e2e 0000 0000 0000 0000  unknown.........
```

这里是从0面0道的第3个扇区倾印片断：

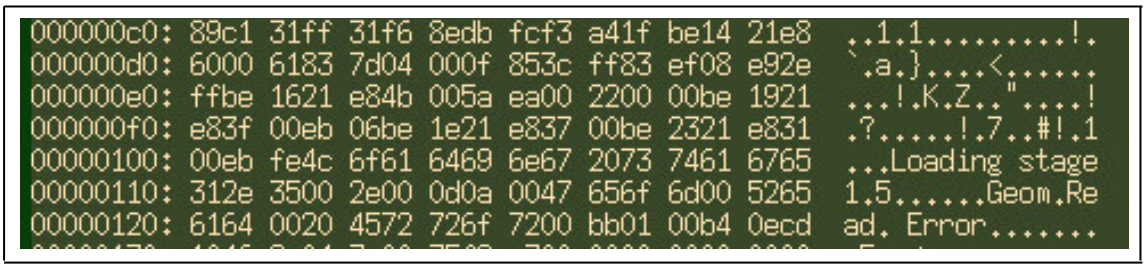
```
00000000: ea70 2200 0000 0302 ffff ff00 0000 0000  .p".............
00000010: 0200 302e 3934 00ff ff09 ff2f 6772 7562  ..0.94...../grub
00000020: 2f73 7461 6765 3220 2f67 7275 622f 6d65  /stage2 /grub/me
00000030: 6e75 2e6c 7374 0000 0000 0000 0000 0000  nu.lst..........
```



从中可见，e2fs_stage1_5的确是被放在了0面0道的第3个扇区开始的扇区集里！

stage1加载了 start 后，start 便加载随后的扇区总共10多 K 的扇区到内存，随后执行这些 stage1_5代码，有了 stage1_5，那么识别文件系统中的 stage2文件，就不是什么难事了。

关于 stage1、start、stage1_5的位置关系，我们从下面的倾印片断中，可得知一二；注意，这个片断处于 stage1_5倾引的开头，而它正是 start.S 的代码(从字符串可得知)：



最后得出结论，stage1.S 被放在 0 面 0 道的第 1 扇区，start.S 被放在 0 面 0 道的第 2 扇区，而与 boot 分区相关的文件系统的 xxfs_stage1_5 被放在 0 面 0 道第 3 扇区开始的扇区里，其占据的扇区数目与该 stage1_5 文件的大小有关。

代码:

```
+-----------+
|    stage1 | <-- 0 面 0 道第 1 扇区
+-----------+
|    start  | <-- 0 面 0 道第 2 扇区
+-----------+
| stage1_5 | <-- 0 面 0 道第 3 扇区
|     ...   |
+-----------+
```

而其余的 stage1_5以及 stage2都作为文件被存放在 boot 分区里。

好了。补充完了 stage1以及 stage1_5的执行流程以及位置关系后，就轮到 stage2这个大约 110KB 左右的 mini OS 了，前面 phyma 斑竹为此开了个头，我觉得 phyma 斑竹写得很好，他对于 stage2的内幕有着很深刻的理解。还请继续，中途打扰了，不好意思，请原谅。

Q&A

引用:

> 作者: lyy9505
>
> 我有个问题：
> grub 中的 stage1。5处在硬盘的第二和第三扇区，这样的话，
> 会不会覆盖掉原本的一些放在此处的有用信息；
>
> 如果有，那么，grub 是用什么方法来处理这个问题？

0面0道的所有扇区都是保留的，BIOS 不会放置任何数据在此。
注意，第一个主分区也是从1面0道的第1扇区开始的。

一、**phyma**:

呵呵，谢谢 home_king 兄弟的补充
但是 stage1.5 确实是不能固定在某某地方的，会根据情况处理（lilo 的处理办法）
参见 stage2/builtins.c 中间的 static int install_func（）

代码:

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

/* If INSTALLADDR is not specified explicitly in the command-line,
determine it by the Stage 2 id. */
```

```
if (! installaddr)
{
      if (! is_stage1_5)
         /* Stage 2. */
         installaddr = 0x8000;
      else
          /* Stage 1.5. */
          installaddr = 0x2000;
 }

*((unsigned long *) (stage1_buffer + STAGE1_STAGE2_SECTOR))
 = stage2_first_sector;
*((unsigned short *) (stage1_buffer + STAGE1_STAGE2_ADDRESS))
 = installaddr;
*((unsigned short *) (stage1_buffer + STAGE1_STAGE2_SEGMENT))
 = installaddr >> 4;
```

因为 grub 需要不只是安装到 mbr，而且需要安装到某个 partition.

二、mrbanana:

以下是我毕业论文的一部分：）对 grub 源代码进行了一些分析，其实也没有什么新的东西，基本上是对斑竹的一个总结，然后细化了一些内容。可能有些不对的地方还请各位指正。

3 GRUB 整体分析

总体上我们可以把 GRUB 看成一个微型的操作系统，他有 Shell，支持 Script，有文件系统……我们可以把 Stage1 和 Stage1.5 看成一个引导程序，而 Stage2 则是一个操作系统，只不过这个操作系统是专门用来引导其他操作系统的操作系统，为此，Stage2 支持像 kernel，initrd，chainloader 等等为此目的而设置的内部“命令”。

3.1 GRUB 引导操作系统的两种方式

3.1.1 直接引导方式

GRUB 同时支持 Linux, FreeBSD, NetBSD 和 OpenBSD。如果想要启动其他的操作系统，你必须使用链式启动方式来启动他们[6]。

通常，GRUB 直接引导操作系统的步骤如下：

(1) 通过'root'指令来设置 GRUB 的主设备指向操作系统映像文件所存储的地方。

(2) 通过'kernel'命令来载入该操作系统的核心映像。

(3) 如果需要模块，通过'module'命令来加载模块。

(4) 运行命令'boot'。

Linux, FreeBSD, NetBSD 和 OpenBSD 使用相同的方式启动。你可以通过'kernel'命令来装载核心映像，然后运行'boot'命令。如果核心需要一些参数的话，只要在'kernal'命令以后追加就可以了。

3.1.2 链式引导方式

如果你要启动一个不被 GRUB 直接支持的操作系统（例如: Windows 95），可以通过链式引导启动一个操作系统。通常来说，那个引导程序和所要启动的操作系统是安装在一个分区中

的。

主要步骤如下：

(1) 通过'rootnoverify'命令设置 GRUB 的主设备指向一个扇区。

grub> rootnoverify (hd0,0)

(2) 通过'makeactive'命令来设置在扇区上的'active'标志位。

grub> makeactive

(3) 通过'chainloader'命令来加载引导程序。

grub> chainloader +1

'+1'表明 GRUB 需要从起始分区读一个扇区。

(4) 运行命令'boot'。

3.2 GRUB 引导操作系统的简要流程分析

3.2.1 从计算机启动到 GRUB 启动操作系统

(1) BIOS 执行 INT 0x19，加载 MBR 至 0x7c00 并跳转执行。如果你安装 GRUB 到 MBR，GRUB 的安装程序会把 Stage1（512B）拷贝到 MBR。视 Stage2 的大小，安装程序会在 Stage1 中嵌入 Stage1_5 或者 Stage2 的磁盘位置信息。

(2) Stage1 开始执行，它在进行直接加载 Stage1_5 或者 Stage2 并跳转执行。不论是哪种情况，这一步的结果都是 Stage2 开始运行了。

(3) Stage2 这个小型的操作系统终于开始正式运行了！它会把系统切入保护模式，设置好 C 运行环境（主要是 BSS）。他会先找 config 文件（就是我们的 menulist），如果没有的话就执行一个 Shell，等待我们输入命令。然后 Grub 的工作就是输入命令-解析命令-执行命令的循环，当然 Stage2 本身是为加载其他操作系统而存在的，所以如果情况允许，在他执行 Boot 命令以后就会把控制权转交出去。

3.2.2 GRUB 的主要启动模块

GRUB 包含如下几个启动模块：两个必须的场景文件，一个叫"Stage 1.5"的可选的场景文件以及 2 个网络启动的映像文件。首先对他们有一个大致的了解。

（下面这几段作者都是翻译自 GRUB 的手册，要详细了解可以直接看章节“10 GRUB image files”——合集者 tbfly）

Stage1

这是一个基本必须的用来启动 GRUB 的映像文件。通常，这个文件是被装载到 MBR 或者启动扇区所在的分区。由于 PC 的启动扇区的大小为 512 字节，所以这个映像文件编译以后也必须为 512 字节。

Stage1 的全部的工作是从本地磁盘把 Stage 2 或者 Stage 1.5 装载进来。由于对 stage1 大小的限制，它通过分程序表的形式来编码 Stage 2 或者 Stage 1.5 的位置，所以在 stage1 是不能识别任何文件系统的。

Stage2

这是 GRUB 的核心映像。它几乎做了除启动它本身以外的所有事情。通常，它被存放为某一种文件系统下，但并非是必须的。

e2fs_stage1_5

fat_stage1_5
ffs_stage1_5
jfs_stage1_5
minix_stage1_5
reiserfs_stage1_5
vstafs_stage1_5
xfs_stage1_5
这些文件被称为stage 1.5，它存在的目的是做为 stage1 与 stage2 之间的桥梁，也就是说，stage1 载入 stage1.5,然后 stage1.5 载入 stage2。
stage1 与 stage1.5 之间的区别是，前者是不识别任何文件系统的但后者识别文件系统（例如 'e2fs_stage1_5' 识别 ext2fs）。所以你可以安全的移动 stage2 的位置，即使是在 GRUB 安装完以后。

nbgrub
这是一个网络启动的映像文件，被类似于以太网启动装载器所使用。它很类似于 stage2，但它还要建立网络，然后通过网络来载入配置文件[7]。

pxegrub
这是另一个网络启动的映像文件。
除了格式以外，它和'nbgrub'是一致的。

4 STAGE1 模块分析
Stage1 模块是整个引导程序的引导模块，是从开机过渡到 GRUB 的第一个模块。Stage1 的代码文件，是源码目录下 stage1/stage1.S，汇编后便成了一个 512 字节的 img，被写在硬盘的 0 面 0 道第 1 扇区，作为硬盘的主引导扇区。
4.1 Stage1.h 文件分析
在此文件中主要是定义了一些在 Stage1.S 文件中使用到的一些常量。
关于这些常量的分析如下：

```
/*  定义了 grub 的版本号，在 stage1 中可以识别他们.*/
#define COMPAT_VERSION_MAJOR 3
#define COMPAT_VERSION_MINOR 2
#define COMPAT_VERSION ((COMPAT_VERSION_MINOR << 8) \
 | COMPAT_VERSION_MAJOR)

/* MBR 最后两个字节的标志*/
#define STAGE1_SIGNATURE 0xaa55

/* BPB (BIOS 参数块 BIOS Parameter Block)的结束标记的偏移，他含有对驱动器的低级参数的说明. */
#define STAGE1_BPBEND 0x3e

/*  主版本号的标记的偏移*/
#define STAGE1_VER_MAJ_OFFS 0x3e
```

```
/* Stage1 启动驱动器的标记的偏移*/
#define STAGE1_BOOT_DRIVE 0x40

/*  强迫使用 LBA 方式的标记的偏移*/
#define STAGE1_FORCE_LBA 0x41

/* Stage2 地址标记的偏移*/
#define STAGE1_STAGE2_ADDRESS 0x42

/* STAGE2 扇区的标记的偏移*/
#define STAGE1_STAGE2_SECTOR 0x44

/* STAGE2_段的标记的偏移*/
#define STAGE1_STAGE2_SEGMENT 0x48

/*  使用 Windows NT 的魔术头标识的偏移*/
#define STAGE1_WINDOWS_NT_MAGIC 0x1b8

/*  分区表起始地址的标记的偏移*/
#define STAGE1_PARTSTART 0x1be

/*  分区表结束地址的标记的偏移*/
#define STAGE1_PARTEND 0x1fe

/* Stage1 堆栈段的起始地址*/
#define STAGE1_STACKSEG  0x2000

/*  磁盘缓冲段。磁盘缓冲必须是 32K 长而且不能跨越 64K 的边界。*/
#define STAGE1_BUFFERSEG 0x7000

/*  驱动器参数的地址*/
#define STAGE1_DRP_ADDR 0x7f00

/*  驱动器参数的大小*/
#define STAGE1_DRP_SIZE  0x42

/*在 BOIS 中软盘的驱动器号标志*/
#define STAGE1_BIOS_HD_FLAG 0x80
```

4.2 Stage1.s 文件分析

首先在这个文件的开始部分定义了一些宏。

```
#define ABS(x) (x-_start+0x7c00)
```

这个宏计算了直接地址。由于 MBR 是被加载到 0x7c00 的位置，所以通过计算可以直接得

到 x 参数的直接地址。这样就可以不依赖于 Linker 程序。

#define MSG(x) movw $ABS(x), %si;
这个宏用于处理对字符串的载入和响应。

然后程序从_start 程序入口开始执行，此入口在内存中的位置为 CS:IP 0:0x7c00。随后对一系列的变量进行了初始化。设置了起始的扇区、磁道和柱面，并设置了他们的起始位置。同时还设置了 stage1 的版本号。通过设置 boot_drive 变量，来设置从那个盘来载入 stage2。如果此变量设置成 0xff 则从默认的启动驱动器中来载入 stage2。然后指定了 stage2 的起始地址是 0x8000，起始段是 0x800，起始的扇区号是 1。也就是说 stage2 起始位置是被存放在 0 柱面，0 磁道，第 2 扇区上的 [8]。
程序从 real_start 入口开始真正执行。首先设置了数据段以及堆栈段的偏移为 0，然后设置 stage1 的堆栈的起始地址为 STAGE1_STACKSEG 即 0x2000。随后打开中断。然后检查是否设置了启动的磁盘。即 boot_drive 变量是否为 0xff，如果非 0xff 则保存设置的磁盘号到 dl 寄存器中，并压入堆栈保存。同时在屏幕上显示 GRUB 字样。然后检查此启动磁盘是否是软盘，如果是软盘则直接跳转到 CHS 模式不用检测是否支持 LBA 模式。然后检测所启动的磁盘是否支持 LBA 模式。接着程序分成两块，一块是 LBA 模式，一块是 CHS 模式。
LBA 英文全名为 Logical Block Addressing，中文名称为逻辑区块寻址[9]。LBA 所指的是一种磁盘设备的寻址技术，它是利用逻辑映对的方式来指定磁盘驱动器的扇区，目前个人 计算机所使用的传输接口中，增强型 IDE (Enhanced IDE) 和 SCSI 均使用逻辑区块寻址方式。传统的硬盘寻址技术是采取实体寻址（physical mapping、physical addressing）的方式，以磁盘上的实际结构，直接作为资料区块地址的结构。但由于初期在设计实体寻址方式时，硬盘容量只有 5、10、20 MB 等等小容量机种，所以设计出来的最大的寻址能力，只能到 1024 个磁柱（cylinder）、16 个磁头（head）、63 个扇区（sector）。 以每个扇区（sector）512 字节（bytes）计算，实体寻址的方式最多只能使用 512×63×1024×16＝528482304 字节（528MB）的硬盘空间。但是由于磁性储存技术不断的提升，硬盘容量大幅增加的情况之下，这样的限制让使用者必须将硬盘画分为多个区块，使用上非常的不方便。
因此硬件厂商研究出了 LBA 逻辑寻址方式，也就是计算机系统并没有将资料存放地点的相关记录，应对到硬盘上资料实际存放的位置。而是由 IDE 控制电路和 BIOS 负责转换寻址（mapping）资料的记录位置表。经过转换后的记录方式，是将第 1 个磁柱上的第 1 条磁道的第 1 个扇区编号为 0，第二个扇区编号为 1，以此类推……，假设 1 条磁道有 2000 个扇区，那么第 2000 个扇区的编号就是 1999。第 2 条磁道上的第 1 个扇区就是 2000，如此一直线性排列下 去。以逻辑区块的方式来寻址的硬盘，最多可达 16383 磁柱，最大磁头数为 16 个，每轨扇区有 63 区，扇区大小为 512 字节，所支持之硬盘空间为 512×63×16383×16＝8455200768 字节（8.4GB）[10]。
在 ATA的接口规格中，定义了使用 28 位来寻址，因此计算出来，它可以支持到 224×512＝137GB 的容量。不过不幸的，BIOS 并无法配合，它使用 24 位来寻址（也就是 LBA 模式）。所以根本之道，就是改变 BIOS 对中断 13h 的支持，因此后来的 BIOS 就设计了加强版的中断 13h。一口气使用了 64 位来对硬盘做寻址，因此可以支持到 264×512＝9.4TB，相当于 3 万亿倍的 8.4GB[11]。
如果是 LBA 模式下读取，首先对先前定义的磁盘的一些参数进行了定义，为以后调用 INT 13 做准备。使用 INT13 的 0x42 功能，把磁盘内容读到内存中。设置 ah 为 0x42 为功能号，设置 dl 寄存器来设置磁盘号，si 为记录磁盘一系列信息 的地址偏移量，磁盘信息中包括了要读入的柱面号、磁道号以及扇区号。程序然后调用 BOIS INT 13 中断将启动磁盘上的第二扇

区上的内容读到内存中的 STAGE1_BUFFERSEG 处，在 Stage1.h 中定义 STAGE1_BUFFERSEG 为 0x7000。即将第二扇区上的内容读到内存中的 0x7000 处。读入成功的话跳转到 COPY_BUFFER 处，如果读取失败则尝试使用 CHS 模式读入。
与 LBA 模式不同的是，调用 BIOS INT 0x13 中断中的 0x2 号功能，设置 ah 寄存器为 0x2，al 为扇区数，cl 的位 6，位 7 和 ch 组合为磁道号，cl 的 0－5 位为扇区号，dh 为磁头 号，dl 为驱动器号（其中 0x80 为硬盘，0x0 为软驱）。es:bx 为数据缓冲区的地址。但所起的功能与前面提到的 LBA 模式是类似的，也是将第二扇区中的内容读到内存中的 0x7000 处，作为缓存。然后跳转到 COPY_BUFFER 处。
最后调用 COPY_BUFFER 将刚刚读入的扇区转移到 stage2_address。即转移到 0x8000 处。
4.3 Stage1 模块功能综述
由于对 Stage1 文件容量的限制，所以 Stgae1 所做的工作相对来说比较有限。它首先被 BOIS 装载到内存中的 0x7c00 处，然后通过调用 BOIS INT13 中断，把第启动驱动器中第二扇区上的内容读到内存中的 0x7000 处，然后通过调用 COPY_BUFFER 将其转移到了内存中 0x8000 的位 置上。这个被读入的第二扇区上的内容，就是下面将要分析的 Start.s 功能模块 。

5 START 模块分析
从上一章节的分析中我们看到，Stage1 的是完成了一个 MBR 所需要完成的任务，但 GRUB 并没有直接就通过 Stage1 直接载入 GRUB 的内核，而是 通过 Stage1 载入了另一个模块到 0x8000 处。根据对源代码的分析，发现被载入的这个模块就是下面需要分析的第二个模块，即 start.S 模块。

5.1 start.S 模块功能分析
在程序的开始部分，仍然是对程序定义了一些宏。

```
#ifdef STAGE1_5
# define ABS(x) (x-_start+0x2000)
#else
# define ABS(x) (x-_start+0x8000)
#endif
```

可以发现，如果定义了 STAGE1_5 则程序的起始地址是 0x2000，而如果没有定义 STAGE1_5 程序起始的地址正好是 0x8000。所以我判断， 在 Stage1 后载入内存的程序部分就是 Start.s 所编译以后的 512 字节的映象文件。关于 STAGE1_5 的部分暂时先不进行分析，这里暂且跳过。

宏 “#define MSG(x) movw $ABS(x), %si;”的作用是在屏幕上显示字符串。
接着就是程序的入口_start。由于是紧接着 Stage1 被载入内存中的，所以它的起始地址就是 0x8000，并且它仍然将使用 Stage1 模块留下来 的寄存器以及变量等信息。如果设置 STAGE1_5 变量则在屏幕上显示“Loading stage1.5”，如果没有设置这个变量则显示“Loading stage2”。然后读入需要读入的扇区的数目。接着进入一个 bootloop 的循环，如果需要读入的扇区不为 0，则继续循环，直到当需要读入的扇区数目 为 0 时，循环结束。在这个循环中，使用与 Stage1 中的方法相同，判断了驱动器磁盘所支持的读写模式，根据不同的磁盘所支持的不同模式，跳转到相应的部 分去读取磁盘上的扇区到内存中去。如果磁盘支持的是 LBA 模式则跳转到 lba_mode 部分读取相应的扇区，如果磁盘不支持 LBA 模式，则跳转到 chs_mode 部分，通过 CHS 模式来读入把磁盘中的扇区读入到内存中。首先是把读到的扇区读到内存中的 0x7000 处缓存起来，然后通过调用 copy_buffer 子程序，把缓存中的内容复

制到目标地址，即 0x8200 开始的地方。与 Stage1 中的一样，在 Start.s 中也有一个记录地址的数据结构，不同的是在 Stage1 中只有一项，而 Start.S 记录的是一个地址的链表，称为 Blocklist，该链表的结点都记录了一个连续 sectors 的集合。

```
lastlist:
  .word 0
  .word 0
  . = _start + 0x200 - BOOTSEC_LISTSIZE
/*加 0x200 是由于 Start.s 编译完以后也是一个 512 字节的映象文件。*/

  /* 初始化了第一个数据列表*/
blocklist_default_start:
  .long 2 /* 记录了从第 3 个扇区开始*/
blocklist_default_len:
  /* 这个参数记录了需要读取多少个扇区 */
#ifdef STAGE1_5
  .word 0 /* 如果设置了 STAGE1_5 标志，则不读入*/
#else
  .word (STAGE2_SIZE + 511) >> 9 /*读入 Stage2 所占的所有扇区*/
#endif
blocklist_default_seg:
#ifdef STAGE1_5
  .word 0x220 /*如果设置 STAGE1_5 则从 0x220 开始读入*/
#else
  .word 0x820 /*如果没有设置 STAGE1_5 则从 0x820 开始读入*/
#endif

firstlist:
```

当把所有需要读入的扇区都读入以后，程序进入 bootit 子程序块。然后程序进行跳转，如果设置了 STAGE1_5 标志，则跳转到 0x2200 执行，如果没有设置 STAGE1_5 标志，则跳转到 0x8200 处继续执行。

5.2 Start 模块功能综述

通过对 Start.s 文件的分析，我们可以看到。Start 模块主要是做了一件事情，就是把 Stage2 或者 Stage1_5 模块从磁盘装载到内存中。如 果是直接装载 Stage2 的话，是装载在内存的 0x8200 处，如果装载 Stage1_5 的话，是装载在内存的 0x2200 处。

6 GRUB Kernel 模块分析

由于我分析的是 GRUB2 的源代码，从 GRUB2 开始，从 Start 模块载入的是 Grub 的整个 kernel。从官方的说明可以看到，与Grub 相比，最大的差异在于GRUB2 将 Stage1.5以及 Stage2 的功能归并为 GRUB2 的 kernel，并提高了压缩性能；编译生成的 kernel——core.img 只有 24KB 左右，即使对于最普遍的 CHS 读写模式所支持的 0 面 0 道的 64 个扇区(折合 32K 左右)而言，空间是足够放置 GRUB2 的 kernel 的，GRUB 的每个 Stage1.5 都至少在 11K 左右，

而 stage2 则为 110K 左右。

6.1 asm.S 文件分析

在分析了 Start 模块以后，发现如果没有设置 Stage1_5 参数，那么系统已经把 Grub kernel 从磁盘完全装载到了起始地址为 0x8200 开始的内存中。于是我便在源代码中寻找起始地址从 0x8200 开始执行的代码。发现 asm.S 文件就是这样一个符合条件的模块。

首先在这个文件的开始，仍然定义了这样一个宏：

```
#ifdef STAGE1_5
# define ABS(x) ((x) - EXT_C(main) + 0x2200)
#else
# define ABS(x) ((x) - EXT_C(main) + 0x8200)
#endif
```

从这个宏中可以看出，从 Start 模块以后从磁盘转载的应该就是这个文件编译以后的模块。程序的入口是 EXT_C(main)。如果没有定义 STAGE1_5 那么程序的起始地址正是 0x8200 完全符合前面所做的分析。同时由于设置了.code16，整个程序开始仍然时工作在实模式下的。

接着分析 ENTRY（main）这个函数。首先为了保证 main 这个函数如果是 Stage2 的话被装载在 0x8200，如果是 Stage1.5 的话被装载在 0x2200。然后程序执行了一个长跳转。ljmp $0, $ABS(codestart)。

在执行 codestart 代码之前，它对一些变量进行了初始化。设置了如版本号、install_partition、saved_entryno、 stage2_id、force_lba 和 config_file 等。如果是 Stgae1.5 则 config_file 为“/boot/grub /stage2”，如果是 Stage2 则为“boot/grub/menu.lst”。

然后进入 codestart 代码，首先关中断，对断寄存器进行了一些初始化，然后设置了堆栈的起始地址为 STACKOFF 即(0x2000 - 0x10)。

接着程序调用了 real_to_prot 这样一个子功能模块。从实模式转换把程序转换到保护模式下。分析 ENTRY(real_to_prot)子功能， 主要的转换步骤如下：首先程序仍然在实模式下，关中断。接着载入了 GDT 表。然后通过.code32 转到保护模式下，跳转到 protcseg 子功能下，重新装载所有的段寄存器。同时把返回的地址放到 STACKOFF 中，然后获得保护模式下的堆栈地址，把 STACKOFF 压入堆栈中，进行保护。然后返回。

然后程序继续，清空了 bss 段，调用了 init_bios_info 函数，这个函数体是整个 C 语言代码的入口，是 Cmain 函数前的初始化代码。

在 asm.s 文件中，主要是一些汇编代码的函数块，没有 C 语言的代码，于是我在 share.h 中找到了 init_bios_info 的函数定义，从而在 common.c 中找到了 init_bios_info 的代码。

同时在这个文件中还定义了非常多的汇编代码写的函数，这些函数将来会被 C 文件调用。这里先对这些文件进行一下说明，如表 6.1 所示。

表 6.1 asm.s 文件中的汇编函数列表

函数名称	函数作用
stop()	调用 prot_to_real 子函数，从保护模式转换成实模式
hard_stop()	通过反复调用自身，形成一个死循环，起到一个暂停的作用
grub_reboot()	重新启动系统
grub_halt(int no_apm)	暂停系统，利用时钟计时，如果设置 NO_APM 将不使用时钟计时
track_int13(int drive)	追踪 INT13 来操作 I/O 的地址空间
set_int15_handler(void)	建立 INT15 的句柄
unset_int15_handler(void)	重新恢复 INT15 的句柄

set_int13_handler(map) 复制一块数据到驱动器并且建立 INT13 的句柄
chain_stage1(segment,offset,part_table_addr) 启动另一个 stage1 的载入程序
chain_stage2(segment, offset, second_sector) 启动另一个 stage2 的载入程序
real_to_prot () 实模式转换成保护模式
prot_to_rea l() 保护模式转换成实模式
int biosdisk_int13_extensions (int ah, int drive, void *dap) 调用 IBM/MS 扩展 INT13 的功能。
int biosdisk_standard (int ah, int drive, int coff,int hoff, int soff,int nsec, int segment) 调用标准的 INT13 功能
int check_int13_extensions (int drive) 检查磁盘是否支持 LBA 模式
get_diskinfo_int13_extensions (int drive, void *drp) 从参数*drp 返回磁盘驱动的具体结构
int get_diskinfo_standard (int drive, unsigned long *cylinders,unsigned long *heads, unsigned long *sectors) 返回指定磁盘的柱面，磁头以及扇区信息
int get_diskinfo_floppy (int drive, unsigned long *cylinders,unsigned long *heads, unsigned long *sectors) 返回软盘的磁盘的柱面，磁头以及扇区信息
get_code_end() 返回代码末端的地址
get_memsize(i) 返回内存大小，如果 I 为 0 返回常规内存，I 为 1 返回扩展内存
get_eisamemsize() 返回 EISA 的内存分布图
get_rom_config_table() 获得 Rom 配置表的线性地址
int get_vbe_controller_info (struct vbe_controller *controller_ptr) 获得 VBE 控制器的信息
int get_vbe_mode_info (int mode_number, struct vbe_mode *mode_ptr) 获得 VBE 模式信息
int set_vbe_mode (int mode_number) 设置 VBE 模式
linux_boot() 做一些危险的设置，然后跳转到 Linxu 安装的入口代码
multi_boot(int start, int mb_info) 这个函数启动一个核心使用多重启动的标准方法
void console_putchar (int c) 通过这个函数在终端上显示字符
int console_getkey (void) 调用 INT16 从键盘上读取字符
int console_checkkey (void) 检查是否某个键被一直按下去
int console_getxy (void) 调用 INT10 获得光标的位置
void console_gotoxy(int x, int y) 调用 INT10 设置光标的位置
void console_cls (void) 调用 INT10 清空屏幕
int console_setcursor (int on) 调用 INT10 设置光标的类型
getrtsecs() 如果第二个值能被读取，则返回这个值
currticks() 用 Ticks 为单位返回当前时间，一秒约为 18－20 个 Ticks

6.2 common.c 文件分析
在 common.c 文件中我找到了函数 init_bios_info 的实现的代码。分析发现，整个 init_bios_info 文件主要是对 multiboot_info 这个结构进行初始化以及填充。
整个结构体分析如下：

```
struct multiboot_info
{
  /* 多重启动信息的版本号*/
  unsigned long flags;

  /* 可以使用的内存 */
  unsigned long mem_lower;
```

```
  unsigned long mem_upper;

  /*  主分区  */
  unsigned long boot_device;

  /*核心的命令行*/
  unsigned long cmdline;

  /*启动模块的列表*/
  unsigned long mods_count;
  unsigned long mods_addr;

  union
  {
     struct
     {
      /* (a.out)  核心标识表的信息  */
     unsigned long tabsize;
     unsigned long strsize;
     unsigned long addr;
     unsigned long pad;
     }
a;

struct
{
   /* (ELF)  核心标识表的信息*/
   unsigned long num;
   unsigned long size;
   unsigned long addr;
   unsigned long shndx;
}
  e;
}
syms;

/*  内存分布图的缓存  */
unsigned long mmap_length;
unsigned long mmap_addr;

/*  驱动器信息缓存  */
unsigned long drives_length;
unsigned long drives_addr;
```

```
  /* ROM 配置表 */
  unsigned long config_table;

  /* 启动装载器的名称 */
  unsigned long boot_loader_name;

  /* APM 表 */
  unsigned long apm_table;

  /* 视频 */
  unsigned long vbe_control_info;
  unsigned long vbe_mode_info;
  unsigned short vbe_mode;
  unsigned short vbe_interface_seg;
  unsigned short vbe_interface_off;
  unsigned short vbe_interface_len;
};
```

通过调用 asm.S 中的底层功能模块，对这个结构进行初始化以后，直接调用 cmain 函数。

6.3 Stage2.c 文件分析

通过查找，在 Stage2.c 中找到了 cmain 函数，这个应该就是 Stage2 这个小型操作系统的入口了。然后程序就进入一个死循环，整个 Stage2 就在这个死循环中运行。接着调用 reset()函数对 stage2 的内部变量进行初始化。通过 open_preset_menu()函数尝试打开已经设置 好的菜单。如果用户没有设置好菜单，那么将返回 0，如果已经设置好了菜单则不返回 0。如果没有成功打开菜单，那么将通过 grub_open()函数尝试打 开 config_file。Grub 使用内部的文件格式来打开这样一个配置文件，如果仍然打开失败，则跳出整个循环。如果打开成功，则根据打开的情况，即 is_preset 变量的值的情况来判断是从预设菜单读入还是从配置文件读入命令。然后通过把 is_preset 传入 get_line_from_config()函数，将命令读入 cmline 中。然后通过 find_command()函数查找有没有这条命令。

在 Grub 中，保存命令的格式是保存在一个 builtin 的结构体中的。这个结构体在 shared.h 头文件中进行了定义。

```
struct builtin
{
  /* 命令名称，重要，是搜索命令时的依据 */
  char *name;
  /* 命令函数，重要，是搜索匹配后调用的函数 */
  int (*func) (char *, int);
  /* 功能标识 */
  int flags;
  /* 简短帮助信息 */
  char *short_doc;
  /* 完整帮助信息 */
```

```
  char *long_doc;
};
```

整个命令的表的定义如下

```
extern struct builtin *builtin_table[];
```

find_command()函数在 cmdline.c 中定义，它对整个 builtin 表进行遍历，然后比较名称。如果在表中发现了这个命令，则返回指向 当前 builtin 结构的指针。如果没有发现这个命令则返回 0 同时返回一个 errnum。如果成功的找到了一条指令，然后通过调用在 cmdline.c 中 的 skip_to ()函数，获得当前 builtin 指针所指向结构的命令的参数。然后通过(builtin->func) (arg, BUILTIN_MENU)直接调用此命令。最后一直循环，直到没有命令可以取为止。

如果由于前面没有成功的打开预先配置的文件而跳出循环，则通过在 cmdline.c 文件中定义的 enter_cmdline()函数调用来启动命令行。在 enter_cmdline()函数中，进入另一个循环等待接受命令。当通过 get_cmdline()函数接收到命令以后，仍然通过 find_command()函数调用来遍历 builtin 表，如果没有在表中找到输入的指令则返回一个 errnum=ERR_UNRECONGNIZED。如果成功找到了这条指令，同样首先调用在 cmdline.c 中的 skip_to ()函数，获得当前 builtin 指针所指向结构的命令的参数。然后(builtin->func) (arg, BUILTIN_MENU)直接调用此命令。

如果成功的打开了菜单则跳转到 run_menu()函数，这里是 grub 中整个 menu 用户界面的主循环。首先有一个计时器 grub_timout 进行计 时，如果 grub_timeout<0 或者没有设置，那么就强行显示菜单。如果菜单没有显示，则在屏幕上显示“Press `ESC' to enter the menu...”，并进入一个死循环中，当用户按下 ESC 按键，则马上显示菜单。如果超时，那么就直接进入第一个，也就时默认的那个启动项目。如果显示菜 单，则显示所有可以选择的入口。

不论是否显示菜单，最后程序都将跳转到 boot_entry。首先程序先清空了屏幕，然后把光标定于第一行的位置。然后再次进入一个循环。然后如果没有设 置入口则通过调用 get_entry()函数来获取一个默认的入口。然后调用在 cmdline 中的 run_script()函数解释这个入口。run_script()函数对这个入口以后的指令脚本，进行了解析。解析的方式仍然是利用 find_command()函数调用。

6.4 GRUB 部分指令说明

Grub 中所有的预先设置的指令都是在 builtins.c 文件中实现的。比如启动一个 FreeBSD 操作系统，可以输入以下的指令：

```
grub> root (hd0,a)
grub> kernel /boot/loader
grub> boot
```

6.4.1 root 指令

调用 root 指令的函数是在 builtins.c 中的 root_func (char *arg, int flags)函数。第一个参数指定了哪个磁盘驱动器，如 hd0 是指第一块硬盘。第二个参数是分区号。然后在 root_func()中它有调用了 real_root_func (char *arg, int attempt_mount)这个函数，并把参数 arg 传入 real_root_func 中并把 attempt_mount 设置为 1。如果传入的 arg 是 空的，那么就直接使用默认的驱动器。然后调用 set_device()函数，从字符串中提取出驱动器号和分区号。测试如果所填写的驱动器号以及分区号读写 没有问题，那么就在变量 saved_partition 和 saved_drive 中保存读取的这两个数据。然后返回。

这个函数主要的作用是为 GRUB 指定一个根分区。

6.4.2 kernel 指令

调用 kernel 指令的函数是在 builtins.c 文件中 kernel_func (char *arg, int flags)函数。在这个函数中，首先进入一个循环，对传进来的参数进行解析。如果“--type=TYPE”参数被设置了，根据传入的参数设置 suggested_type 变量赋予不用的操作系统的值。当没有别的参数被设置以后，则跳出循环。然后从参数中获得内核的文件路径，赋值给 mb_cmdline 变量，然后通过 load_image()函数载入核心，并且返回核心的类型。如果返回的核心类型是 grub 不支持得类型，即 kernel_type == KERNEL_TYPE_NONE 返回 1，成功则返回 0。
这个函数主要的作用是，载入操作系统的核心。

6.4.3 boot 指令

调用 boot 指令的函数是在 builtins.c 文件中的 boot_func (char *arg, int flags)函数。如果被载入的核心类型不是未知的，那么调用 unset_int15_handler()函数，清除 int15 handler。接着根据 grub 支持的不同的操作系统调用相应的启动程序。当启动的内核为 BSD 时调用 bsd_boot ()函数，当启动的内核为 LINUX 时调用的函数时 linux_boot()函数，当启动方式是链式启动方式时，调用 chain_stage1()函数， 当启动方式是多重启动时，调用 multi_boot()函数。
这个函数主要的作用是，根据不同的核心类型调用相应的启动函数。

6.5 GRUB Kernel 分析总结

通过分析，这个核心模块主要的工作是完成了 GRUB 这个微型操作系统的从磁盘到内存的装载和运行。在 asm.S 这个文件中提供了从汇编代码到 C 代码转换的 接口，也是从这里开始正式载入了 GRUB 这个微型操作系统，可以说是 GRUB 运行的一个入口。同时，在 asm.S 文件中，对底层的方法用汇编语言进行了封 装，方便在以后的 C 代码中调用。然后经过对 BIOS 进行一些初始化以后，正式进入了 GRUB 的主程序，即在 stage2 中的 cmain 入口。从此这个微型 的操作系统开始正式运行。然后值得注意的是 buildin 这个数据结构，这个结构就是 GRUB 所有支持命令的数据结构。结构包括了一个用来识别的名字和一 个用来调用的方法。GRUB 通过接收外部输入的指令的方式，来间接的启动和装载其他的操作系统。

7 总结

7.1 GRUB 源代码分析总结

通过对整个源代码的分析，大致上整个 GRUB 启动到引导其他操作系统分为如下几个步骤。
第一步 开机后，通过 BIOS 装载 Stage1 模块
第二步 通过 Stage1 模块装载 Start 模块
第三步 通过 Start 模块将整个 GRUB 的内核载入内存
第四步 通过 GRUB 的一个 Shell 的机制，作为一个小型的操作系统，来通过指令的方式装载不同的其他操作系统。
整个过程中 GRUB 启动的内存映象图如图 7.1 所示。
总体分析下来，首先感觉到 GRUB 整个代码在编码方面是非常严谨的，特别是整个程序的构架体现出了它灵活容易扩展的特性。主要体现在，它有别于普通的操作 系统引导程序，在 BIOS 启动时就直接去装载特定操作系统的模块或者内核，而是通过 BIOS 的功能首先装载了一个属于自己的引导程序，也可以理解为 GRUB 这个操作系统的引导程序。也就是说在引导任何用户的操作系统之前 GRUB 首先引导的是它的本身。这样为将来的扩展性打下了非常好的基础。

由于 GRUB 采用了类似 Shell 的方式来解释并运行用户设计好的脚本或者接受用户输入的指令，并且为用户提供了非常好的底层的方法接口，所以用户可以非 常灵活的组合指令来引导不同的操作系统，同时并不要求用户对底层的物理结构有非常高的了解，只要能使用提供的指令就可以来操作配置多重启动的多个操作系 统。并且，GRUB 提供了非常好的人机交互界面，可以通过预先设置的指令，或者菜单来显示让用户选择操作系统，相对来说就比较易用。

同时 GRUB 也提供了非常好的扩展性，这个也是由于 GRUB 特殊的结构保证的。首先 GRUB 是一个开源的项目，它的源代码是向所有的用户和开发着公开的， 这样无论是用户的需求发生了变化，还是硬件的标准得到了提升，都能很快的在 GRUB 中得到实现。其次，GRUB 的指令是非常容易添加的，用户只要了解了 GRUB 指令的格式，就能非常容易的在 GRUB 原始指令的基础上添加属于自己的指令，这样的设计相当程度上提高了程序的模块化，耦合度比较低。

<原链接中缺图 note by tbfly>

图 7.1 GRUB 启动内存映像图

三、mips_linux:

还差这一块内容，*_stage1_5 and stage2 是怎么 build 出来的，以下来自于 build 过程，写的相当清楚了。

Let's see how *_stage1_5 and stage2 are generated:

---------- BEGIN ----------

e2fs_stage1_5:

gcc -o e2fs_stage1_5.exec -nostdlib -Wl,-N -Wl,-Ttext -Wl,2000
e2fs_stage1_5_exec-start.o e2fs_stage1_5_exec-asm.o
e2fs_stage1_5_exec-common.o e2fs_stage1_5_exec-char_io.o
e2fs_stage1_5_exec-disk_io.o e2fs_stage1_5_exec-stage1_5.o
e2fs_stage1_5_exec-fsys_ext2fs.o e2fs_stage1_5_exec-bios.o

objcopy -O binary e2fs_stage1_5.exec e2fs_stage1_5

stage2:

gcc -o pre_stage2.exec -nostdlib -Wl,-N -Wl,-Ttext -Wl,8200
pre_stage2_exec-asm.o pre_stage2_exec-bios.o pre_stage2_exec-boot.o
pre_stage2_exec-builtins.o pre_stage2_exec-common.o
pre_stage2_exec-char_io.o pre_stage2_exec-cmdline.o
pre_stage2_exec-disk_io.o pre_stage2_exec-gunzip.o
pre_stage2_exec-fsys_ext2fs.o pre_stage2_exec-fsys_fat.o
pre_stage2_exec-fsys_ffs.o pre_stage2_exec-fsys_minix.o
pre_stage2_exec-fsys_reiserfs.o pre_stage2_exec-fsys_vstafs.o
pre_stage2_exec-hercules.o pre_stage2_exec-serial.o
pre_stage2_exec-smp-imps.o pre_stage2_exec-stage2.o
pre_stage2_exec-md5.o

objcopy -O binary pre_stage2.exec pre_stage2

cat start pre_stage2 > stage2

---------- END ----------

According to the output above, the layout should be:
e2fs_stage1_5:
[start.S] [asm.S] [common.c] [char_io.c] [disk_io.c] [stage1_5.c]
[fsys_ext2fs.c] [bios.c]
stage2:
[start.S] [asm.S] [bios.c] [boot.c] [builtins.c] [common.c] [char_io.c]
[cmdline.c] [disk_io.c] [gunzip.c] [fsys_ext2fs.c] [fsys_fat.c]
[fsys_ffs.c] [fsys_minix.c] [fsys_reiserfs.c] [fsys_vstafs.c]
[hercules.c] [serial.c] [smp-imps.c] [stage2.c] [md5.c]
We can see e2fs_stage1_5 and stage2 are similar. But e2fs_stage1_5 is smaller, which contains basic modules(disk io, string handling, system initialization, ext2/3 file system handling), while stage2 is all-in-one, which contains all file system modules, display, encryption, etc.
start.S is very important for Grub. stage1 will load start.S to 0200:0000(if stage1_5 is configured) or 0800:0000(if not), then jump to it. The task of start.S is simple(only 512Byte),it will load the rest parts of stage1_5 or stage2 to memory. The question is, since the file-system related code hasn't been loaded, how can grub know the location of the rest sectors? start.S makes a trick:

```
------------------------------ BEGIN --------------------------------
blocklist_default_start:
.long 2 /* this is the sector start parameter, in logical
sectors from the start of the disk, sector 0 */
blocklist_default_len: /* this is the number of sectors to read */
#ifdef STAGE1_5
.word 0 /* the command "install" will fill this up */
#else
.word (STAGE2_SIZE + 511) >> 9
#endif
blocklist_default_seg:
#ifdef STAGE1_5
.word 0x220
#else
.word 0x820 /* this is the segment of the starting address
to load the data into */
#endif
firstlist: /* this label has to be after the list data!!! */
-------------------------------- END --------------------------------
```

an example:

```
# hexdump -x -n 512 /boot/grub/stage2
...
00001d0 [ 0000 0000 0000 0000 ][ 0000 0000 0000 0000 ]
00001e0 [ 62c7 0026 0064 1600 ][ 62af 0026 0010 1400 ]
00001f0 [ 6287 0026 0020 1000 ][ 61d0 0026 003f 0820 ]
```

We should interpret(backwards) it as: load 0x3f sectors(start with No. 0x2661d0) to 0x0820:0000, load 0x20 sectors(start with No.0x266287) to 0x1000:0000, load 0x10 sectors(start with No.0x2662af) to 0x1400:00, load 0x64 sectors(start with No.0x2662c7) to 0x1600:0000.
In my distro, stage2 has 0xd4(1+0x3f+0x20+0x10+0x64) sectors, file size is 108328 bytes, the two matches well(sector size is 512).
When start.S finishes running, stage1_5/stage2 is fully loaded. start.S jumps to asm.S and continues to execute.
There still remains a problem, when is stage1.5 configured? In fact, stage1.5 is not necessary. Its task is to load /boot/grub/stage2 to memory. But pay attention, stage1.5 uses file system to load file stage2: It analyzes the dentry, gets stage2's inode, then stage2's blocklists. So if stage1.5 is configured, the stage2 is loaded via file system; if not, stage2 is loaded via both stage2_sector in stage1 and sector lists in start.S of stage2.
To make things clear, suppose the following scenario: (ext2/ext3)
mv /boot/grub/stage2 /boot/grub/stage2.bak
If stage1.5 is configured, the boot fails, stage1.5 can't find /boot/grub/stage2 in the file-system. But if stage1.5 is not configured, the boot succeeds! That's because mv doesn't change stage2's physical layout, so stage2_sector remains the same, also the sector lists in stage2.
Now, stage1 (-> stage1.5) -> stage2. Everything is in position. asm.S will switch to protected mode, open /boot/grub/grub.conf(or menu.lst), get configuration, display menus, and wait for user's interaction. After user chooses the kernel, grub loads the specified kernel image(sometimes ramdisk image also), then boots the kernel.

下面大家在一起分析一下 grub 里用到的一些底层实现吧。
主要在 asm.S 中，比如，
real_to_prot () 实模式转换成保护模式
prot_to_real() 保护模式转换成实模式
从 why--how 2 个角度分析，其实代码里还有很多看起来似乎简单，细琢摸起来很有深度暗藏玄机的地方。
希望大家踊跃跟贴。

四、Icoming

引用:

> 作者: biosedit
>
> stage1要占用512字节，那分区表存放在什么地方呢？

```
+----------------+
| |0000
```

在 stage1.S 中有如下的代码：

```
part_start:
  . = _start + STAGE1_PARTSTART
  ……

  . = _start + STAGE1_PARTEND
```

如果 stage1 是写到 MBR 里的话，这里的代码是不会写到硬盘里去的。（因为这部分刚好是硬盘的分区表）

但是如果是写到软盘上的话，这部分的代码是会被写到磁盘上的

我觉得很奇怪的是，为什么 stage1 要先把 start 加载 0x7000 的位置，再考到 0x8000 的位置？直接到 0x8000 不就好了？

Q&A:

[ASK]falcon_16:

有一个问题：

在 stage2/start.S 中最后有一段变量的定义：

```
blocklist_default_len:
  /* this is the number of sectors to read */
```

```
#ifdef STAGE1_5
  .word 0 /* the command "install" will fill this up */
#else
  .word (STAGE2_SIZE + 511) >> 9
#endif
```

其中的 STAGE2_SIZE 明显应该在 stage2_size.h 中定义，因为在文件头部有：

```
#ifndef STAGE1_5
#include <stage2_size.h>
#endif
```

但是我在 grub-0.93 下面没有找到 stage2_size.h 这个文件，按说它应该在 stage2 下面，谁知道它在什么地方吗？

[ANSWER]vramigo:

stage2_size.h 文件是在编译过程中，根据 stage2 映象的具体文件大小来设置的，具体可以参看一下 stage2 目录下的 Makefile.am 文件中的脚本语句：

常量 STAGE2_SIZE 是在编译时根据 stage2 映象的具体大小自动生成的，具体可以参看 stage2/Makefile.am:

```
set dummy `ls -l pre_stage2`;\
echo "#define STAGE2_SIZE $6" > stage2_size.h
```

你可以实际试一下，执行以上两条脚本语句，确实可以得到 stage2_size.h 文件，且里面定义的是常量 STAGE2_SIZE 的大小。

五、cnlas:

那我不自量力说说 Grub 中实模式和保护模式的切换把。。。==

其实 Grub 的实现方法很标准（也许有什么巧妙的地方我没看出来？=v=b）

才疏学浅。。。望各位前辈指教。。。

PS：因为本来是写给学校的同学看的。。。所以有些地方说的比较基础。。。==

标题：Grub 中实现实模式和保护模式切换部分的代码分析

作者：CNLAS

说明：都是些个人愚见。。。出错勿怪。。。还请各位大牛指出。。。==

/* */中为原文件的注释

/*==*/中为我加的注释

源文件位于\stage2\asm.S 版本为 grub-0.97

提示：对 win32asm 爱好者一点小提示，grub 的 asm 部分编译是由 gcc 中的 gas 来完成的。。。gas 采用的 asm 是延续 UNIX 的 AT&T 格式。。。不太了解的同学可以先 google 学习一下。。。XD

```
/*
```

```
 * These next two routines, "real_to_prot" and "prot_to_real" are structured
 * in a very specific way. Be very careful when changing them.
 *
 * NOTE: Use of either one messes up %eax and %ebp.
 */
ENTRY(real_to_prot)
 .code16
/*=在 Grub 这个保护模式过程中。。。只加载了 GDT。。。没有设定 IDT。。。所以这个保护模式要在关闭可屏蔽中断的情况下运行=*/
 cli
 /* load the GDT register */
/*=向 gdtr 寄存器中写入 GDT=注1=*/
 DATA32 ADDR32 lgdt gdtdesc
 /* turn on protected mode */
/*=修改 cr0中的 pe 位切换到保护模式， CR0_PE_ON 在 shared.h 中定义为0x1。。。即将 PE 位置1=*/
 movl %cr0, %eax
 orl $CR0_PE_ON, %eax
 movl %eax, %cr0
 /* jump to relocation, flush prefetch queue, and reload %cs */
 DATA32 ljmp $PROT_MODE_CSEG, $protcseg
 /*
 * The ".code32" directive only works in GAS, the GNU assembler!
 * This gets out of "16-bit" mode.
 */
 .code32
protcseg:
 /* reload other segment registers */
/*=保护模式和实模式中段寄存器的作用是不同的。。。所以这里加载的是段选择子（descriptor）PROT_MODE_DSEG 在 shared.h 中定义为0x10。。。即 GDT 中保护模式下数据段的选择子=*/
 movw $PROT_MODE_DSEG, %ax
 movw %ax, %ds
 movw %ax, %es
 movw %ax, %fs
 movw %ax, %gs
 movw %ax, %ss
 /* put the return address in a known safe location */
/*=保存返回地址。。。STACKOFF 在 shared.h 中定义为(0x2000 - 0x10)=*/
 movl (%esp), %eax
 movl %eax, STACKOFF
 /* get protected mode stack */
/*=建立保护模式下的栈。。。注 2=*/
 movl protstack, %eax
```

```
  movl %eax, %esp
  movl %eax, %ebp
  /* get return address onto the right stack */
/*=返回地址入栈=*/
  movl STACKOFF, %eax
  movl %eax, (%esp)
  /* zero %eax */
  xorl %eax, %eax
  /* return on the old (or initialized) stack! */
  ret

ENTRY(prot_to_real)
  /* just in case, set GDT */
  lgdt gdtdesc
  /* save the protected mode stack */
/*=保存保护模式下的栈以便下次进入时再次使用=*/
  movl %esp, %eax
  movl %eax, protstack
  /* get the return address */
  movl (%esp), %eax
  movl %eax, STACKOFF
  /* set up new stack */
  movl $STACKOFF, %eax
  movl %eax, %esp
  movl %eax, %ebp
  /* set up segment limits */
/*=装入 GDT 中实模式下数据段的选择子。。。PSEUDO_RM_DSEG 是 GDT 中一个规范段描
述符。。。具体意义后面细讲。。。注3=*/
  movw $PSEUDO_RM_DSEG, %ax
  movw %ax, %ds
  movw %ax, %es
  movw %ax, %fs
  movw %ax, %gs
  movw %ax, %ss
  /* this might be an extra step */
  ljmp $PSEUDO_RM_CSEG, $tmpcseg /* jump to a 16 bit segment */
tmpcseg:
  .code16
  /* clear the PE bit of CR0 */
  /*=修改 cr0的 pe 位。。。返回实模式。。。CR0_PE_OFF 在 shared.h 中定义为0xfffffffe。。。即
将 PE 位置0=*/
  movl %cr0, %eax
  andl $CR0_PE_OFF, %eax
  movl %eax, %cr0
```

```
  /* flush prefetch queue, reload %cs */
  DATA32 ljmp $0, $realcseg
realcseg:
  /* we are in real mode now
  * set up the real mode segment registers : DS, SS, ES
  */
  /* zero %eax */
/*=这里才是真正将实模式下数据段装入段寄存器=*/
  xorl %eax, %eax
  movw %ax, %ds
  movw %ax, %es
  movw %ax, %fs
  movw %ax, %gs
  movw %ax, %ss
  /* restore interrupts */
/*=已经回到实模式了。。。置 IF 为1开启可屏蔽中断。。。中断响应交由中断向量表来控制=*/
  sti
  /* return on new stack! */
  DATA32 ret
  .code32
```

注1：

```
DATA32 ADDR32 lgdt gdtdesc
```

gdtdesc 是在 asm.S 最后定义的 GDT 选择子。。。由 lgdt 命令读入。。。该命令将48位存储器操作数读入 gdtr 寄存器。。。高双字为段基地址。。。低字是以字节为单位的段界限

```
/* this is the GDT descriptor */
gdtdesc:
  .word 0x27 /* limit */ /*=定义了5个表项。。。每项8位。。共40位 。。。=*/
  .long gdt /* addr */
```

gdt 的定义就在 gdtdesc 的上方。。。而且源代码中还很友好的给出了 GDT 表项的结构图

```
/*
  * This is the Global Descriptor Table
  *
  * An entry, a "Segment Descriptor", looks like this:
  *
  * 31 24 19 16 7 0
  * ------------------------------------------------------
  * | | |B| |A| | | |1|0|E|W|A| |
  * | BASE 31..24 |G|/|0|V| LIMIT |P|DPL| TYPE | BASE 23:16 |
```

```
 * | | |D| |L| 19..16| | |1|1|C|R|A| |
 * ----------------------------------------------------
 * | | |
 * | BASE 15..0 | LIMIT 15..0 |
 * | | |
 * ----------------------------------------------------
 *
 * Note the ordering of the data items is reversed from the above
 * description.
 */

 .p2align 2 /* force 4-byte alignment */
gdt:

/*=GDT 的0项。。。必须为空=*/
 .word 0, 0
 .byte 0, 0, 0, 0

/*=保护模式下代码段的描述符
由表项定义它是一个可执行可读非一致的有效的32位代码段。。。
基地址是00000000H。。。以 4K 字节为单位的段界限值是0FFFFFH
描述符特权级 DPL=0
=*/
 /* code segment */
 .word 0xFFFF, 0
 .byte 0, 0x9A, 0xCF, 0

/*=保护模式下数据段的描述符
由表项定义它是一个可读可写有效的32位数据段。。。
基地址是00000000H。。。以 4K 字节为单位的段界限值是0FFFFFH
描述符特权级 DPL=0
=*/
 /* data segment */
 .word 0xFFFF, 0
 .byte 0, 0x92, 0xCF, 0

/*=实模式下代码段的描述符
由表项定义它是一个可执行可读一致有效的16位代码段。。。
基地址是00000000H。。。以字节为单位的段界限值是0FFFFH
描述符特权级 DPL=0
=*/
 /* 16 bit real mode CS */
 .word 0xFFFF, 0
 .byte 0, 0x9E, 0, 0
```

```
/*=实模式下数据段的描述符
由表项定义它是一个可读可写有效的16位数据段。。。
基地址是00000000H。。。以字节为单位的段界限值是0FFFFH
描述符特权级 DPL=0
=*/
  /* 16 bit real mode DS */
  .word 0xFFFF, 0
  .byte 0, 0x92, 0, 0
```

注2：

```
movl protstack, %eax
```

1）protstact 是在 asm.S 中定义个一个长整型。。。初始化值为 PROTSTACKINIT

```
protstack:
  .long PROTSTACKINIT
```

2）PROTSTACKINIT 在 shared.h 中定义为(FSYS_BUF - 0x10)

```
#define PROTSTACKINIT (FSYS_BUF - 0x10)
```

3）FSYS_BUF 在 shared.h 中定义为

```
#define FSYS_BUF RAW_ADDR (0x68000)
```

4）RAW_ADDR 在 shared.h 的最上面是这么定义的。。。

```
/* Maybe redirect memory requests through grub_scratch_mem. */
#ifdef GRUB_UTIL
extern char *grub_scratch_mem;
# define RAW_ADDR(x) ((x) + (int) grub_scratch_mem)
# define RAW_SEG(x) (RAW_ADDR ((x) << 4) >> 4)
#else
# define RAW_ADDR(x) (x)
# define RAW_SEG(x) (x)
#endif
```

注3：

为何要在切换会实模式之前把一个规范化描述符选择子装入那些段寄存器呢？

这涉及到80x86架构的 CPU 内部结构问题。。。

从80286开始每个段寄存器都配有一个高速缓冲寄存器。。。。这些寄存器对程序员是不可见的。。。。当给段寄存器装入选择子的时候，处理器自动从描述符 表中将对应的描述符信息装入与段寄存器对应的高速缓冲寄存器中。。。以后访问该段时只要从这里读信息就行了。。。不必再去描述符表中查询。。。在保护模式 下。。。可以通过将相应的选择子装入段寄存器来刷新这些高速缓冲器

在实模式中这些高速缓冲寄存器仍然发挥作用。。。实模式中段基址是段值*16。。。每个段的 32 位段界限都是固定的0FFFFH。。。而且很多属性都是固 定的。。。实模式中不存在 GDT 没有办法设置这些高速缓冲寄存器中的属性。。。只能沿用保护模式下的值。。。所以在从保护模式返回实模式之前。。。一定要 加载一个合适的描述符选择子到相关的段寄存器。。。从而使得

这些高速缓冲寄存器满足实模式的要求

实力所限。。。只能分析到这种地步了。。。望达人修正补充。。。多谢~~

参考链接：

1. http://www.gnu.org/software/grub/manual/grub.html
2. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
3.