MiniOS使用GRUB引导的说明文档

作者：孙东

版本：V0.2

时间：2023-4-1

一、GRUB简介

GNU GRUB（简称“GRUB”）是一个来自GNU项目的启动引导程序。GRUB允许用户可以在计算机内同时拥有多个操作系统，并在计算机启动时选择希望运行的操作系统。它工作在BIOS自检之后，kernel（或者os自己的boot）启动之前，其主要功能就是加载kernel（或者os自己的boot）进内存然后执行它。GRUB目前支持三种启动规范：chainloader（链式引导）、multiboot（多重引导）、Linux专用引导；Linux专用引导是所有以Linux为内核发行版本的操作系统专用的一种引导方式，chainloader、multiboot引导可以完成一些非Linux操作系统的引导功能。chainloader、multiboot的详细介绍如下：

1.1 GRUB chainloader

chainloader的主要作用是**将os自己的引导程序加载进内存，然后将控制权转交给OS 引导程序。**

一个使用链式引导的OS具体引导过程如下`BIOS加载MBR到0x7c00处然后执行MBR --> MBR读取分区表然后将活动分区的Boot加载进内存然后执行它 -->Boot在活动分区中寻找loader程序然后加载loader程序到内存然后执行它 --> loader加载kernel程序到内存然后正式进入OS`。GRUB chainloader取代了`MBR读取分区表然后将活动分区的Boot加载进内存然后执行它`这个过程，该过程中涉及到两个关键问题：

- boot将会被加载到哪。活动分区的boot程序将会被加载到物理地址为**0x7c00**的内存中。GRUB会将boot程序（活动分区的0号扇区）加载到0x7c00处，该操作会覆盖掉BIOS加载的MBR，但是并不会造成影响，原因是：GRUB程序本身也是分段加载的。安装有GRUB的硬盘的第0号物理扇区（MBR）的存储的是GRUB的加载程序，该加载程序会在硬盘中加载GRUB的主程序，然后执行GRUB；当GRUB开始执行之后，0x7c00处的加载程序就不会再被用到。
- GRUB chainloader 如何向boot传递参数。首先GRUB chainloader向活动分区的boot程序传递的参数为该活动分区的分区表表项以及硬盘编号。

  分区表表项的结构如下图所示：

偏移	长度	描述
0	1	状态（80h=可引导，00h=不可引导，其他=不合法）
1	1	起始磁头号
2	1	起始扇区号（仅用了低6位，高2位为起始柱面号的第8,9位）
3	1	起始柱面号的低8位
4	1	分区类型（System ID）
5	1	结束磁头号
6	1	结束扇区号（仅用了低6位，高2位为结束柱面号的第8,9位）
7	1	结束柱面号的低8位
8	4	起始扇区的LBA
12	4	扇区数目

chainloader加载完活动分区的boot程序之后会执行boot程序，**执行时各个寄存器的初始状态**如下：

```
struct grub_relocator16_state state = {
  .edx = boot_drive, //硬盘设备编号
  .esi = boot_part_addr,//启动分区的分区表表项的地址
  .ds = 0,
  .es = 0,
  .fs = 0,
  .gs = 0,
  .ss = 0,
  .cs = 0,
  .sp = GRUB_MEMORY_MACHINE_BOOT_LOADER_ADDR,//boot的加载地址 0x7c00
  .ip = GRUB_MEMORY_MACHINE_BOOT_LOADER_ADDR,//boot的加载地址 0x7c00
  .a20 = 0
};
```

其中*DS:SI*处为启动分区的分区表表项的地址，*DL*寄存器中为硬盘设备在BIOS中的编号。

1.2 GRUB multiboot

GRUB multiboot 与chainloader相比，multiboot引导方式更加**“直接”**。multiboot可以直接将分区中的kernel.elf文件加载进内存，并执行它。要想使用multiboot直接加载kernel并执行它，kernel.elf镜像文件除了遵循ELF文件的头部信息之外还应当遵循multiboot规范（multiboot header）。

1.2.1 multiboot header的布局

multiboot header通常以结构体的形式被编译在代码段的开头，各成员信息如下：

Offset	Type	Field Name	Note
0	u32	magic	required
4	u32	flags	required
8	u32	checksum	required
12	u32	header_addr	if flags[16] is set
16	u32	load_addr	if flags[16] is set
20	u32	load_end_addr	if flags[16] is set
24	u32	bss_end_addr	if flags[16] is set
28	u32	entry_addr	if flags[16] is set
32	u32	mode_type	if flags[2] is set
36	u32	width	if flags[2] is set
40	u32	height	if flags[2] is set
44	u32	depth	if flags[2] is set

magic 、flags 、checksum三个值是必须要有的。multiboot header的各个成员变量的含义解释如下：

- Magic
  Magic 域是标志头的魔数, 它必须等于十六进制值 0x1BADB002。
- Flags
  Flags 域指出 OS 映像需要引导程序提供或支持的特性。
  0-15 位指出需求：如果引导程序发现某些值被设置但出于某种原因不理解或不能不能满足相应的需求, 它必须告知用户并宣告引导失败。
  16-31 位指出可选的特性：如果引导程序不能支持某些位, 它可以简单的忽略它们并正常引导. 所有 flags 字中尚未定义的位必须被置为 0. 这样,flags 域既可以用于版本控制也可以用于简单的特性选择。
- Checksum
  Checksum 域 checksum 是一个 32 位的无符号值, 当与其他的 magic 域（也就是 magic 和 flags）相加时, 结果必须是 32 位的无符号值 0（即 magic + flags + checksum = 0）。
- The address fields of Multiboot header
  所有由 flags 的第 16 位开启的地址域都是物理地址. 它们的意义如下:
- header_addr
  包含对应于 Multiboot 头的开始处的地址——这也是 magic 值的物理地址. 这个域用来同步 OS 映象偏移量和物理内存之间的映射。
- load_addr
  包含 text 段开始处的物理地址. 从 OS 映象文件中的多大偏移开始载入由头位置的偏移量定义, 相减（header_addr - load_addr）.load_addr 必须小于等于 header_addr。
- load_end_addr, 包含 data 段结束处的物理地址
  （load_end_addr - load_addr）指出了引导程序要载入多少数据. 这暗示了 text 和 data 段必须在

OS 映象中连续；现有的 a.out 可执行格式满足这个条件. 如果这个域为 0, 引导程序假定 text 和 data 段占据整个 OS 映象文件。
- bss_end_addr, 包含 bss 段结束处的物理地址
  引导程序将这个区域初始化为 0, 并保留这个区域以免将引导模块和其他的于查系统相关的数据放到这里. 如果这个域为 0, 引导程序假定没有 bss 段。
- entry_addr 操作系统的入口点, 引导程序最后将跳转到那里。

1.2.2 multiboot执行kernel时各寄存器的状态值

- EAX 必须包含魔数 0x2BADB002；
- EBX 必须包含由引导程序提供的 Multiboot 信息结构的物理地址.
- CS 必须是一个偏移量位于 0 到 0xFFFFFFFF 之间的 32 位可读 / 可执行代码段. 这里的精确值未定义.
- Others register(DS,ES,FS,GS,SS), 必须是一个偏移量位于 0 到 0xFFFFFFFF 之间的 32 位可读 / 可执行代码段. 这里的精确值未定义.
- A20 gate, 必须已经开启.
- CR0 第 31 位（PG）必须为 0. 第 0 位（PE）必须为 1. **关闭页式管理，开启保护模式.**
- EFLAGS 第 17 位（VM）必须为 0. 第 9 位（IF）必须为 1. 其他位未定义. 所有其他的处理器寄存器和标志位未定义. 这包括：
- ESP 当需要使用堆栈时,OS 映象必须自己创建一个.
- GDTR 尽管段寄存器像上面那样定义了,**GDTR 也可能是无效的, 所以 kernel一开始决不能载入任何段寄存器（即使是载入相同的值也不行！）直到它设定了自己的 GDT。**
- IDTR **kernel必须在设置完它的 IDT 之后才能开中断.**

在进入kernel时，EBX 寄存器包含**Multiboot 信息数据结构的物理地址**, multiboot引导程序通过它将重要的引导信息传递给kernel。 kernel可以按自己的需要使用或者忽略任何部分；所有的引导程序传递的信息只是建议性的。Multiboot 信息结构和它的相关的子结构可以由引导程序放在任何位置（当然, 除了保留给内核和引导模块的区域）。

注意：

- multiboot跳转到kernel的入口开始执行kernel代码段的时候，此时保护模式已经被打开，但是GDTR寄存器的值可能并不是GDT的地址，因此kernel在更改段寄存的值之前必须重新构建GDT，此处建议进入kernel的第一件事就是重新初始化GDT。
- IDTR寄存器的值也并非IDT的地址，因此kernel在重新初始化IDT之前也不要执行开中断操作。

1.2.3 multiboot 信息结构的格式

multiboot 信息结构是multiboot向kernel传递的一些参数，这些参数记录着当前机器的一些状态信息，其具体格式如下：

```
typedef struct multiboot_t {
  /* multiboot version info 且是必须的 */
  uint32_t flags;

  /* 如果 flags[0] 被置位则出现
   * mem_lower 和 mem_upper 分别指出低端和高端内存的大小, 单位是 k
   * 低端内存的首地址是 0, 高端内存首地址是 1M
   * 低端内存的最大值可能是 640k
   * 高端内存的最大值可能是最大值 -1M, 但并不保证是这个值.
   */
  uint32_t mem_lower;
  uint32_t mem_upper;
```

```
/* 如果 flags[1] 被置位则出现，并指出引导程序从哪个 BIOS 磁盘设备载入的 OS 映像.
 * 如果 OS 映像不是从一个 BIOS 磁盘载入的，这个域就决不能出现 (第 3 位必须是 0).
 * 操作系统可以使用这个域来帮助确定它的 root 设备，但并不一定要这样做.
 */
uint32_t boot_device;

/* 如果 flags[2] 被置位则出现，如果设置了 flags longword 的第 2 位，
 * 则 cmdline 域有效，并包含要传送给内核的命令行参数的物理地址.
 * 命令行参数是一个正常 C 风格的以 0 终止的字符串.
 */
uint32_t cmdline;

/* 如果 flags[3] 被置位则出现，则 mods 域指出了同内核一同载入的有哪些引导模块，
 * 以及在哪里能找到它们.mods_count 包含了载入的模块的个数，
 * mods_addr 包含了第一个模块结构的物理地址.
 */
uint32_t mods_count;
uint32_t mods_addr;

/* offset 28-30 syms
 * 如果 flags[4] 或 flags[5] 被置位则出现 (互斥)，这里是 5 被置位.
 * ELF format section head，参见 i386 ELF 文档以得到如何读取 section 头的更多的细节
 */
uint32_t num;
uint32_t size;
uint32_t addr;
uint32_t shndx;

/* 如果 flags[6] 被置位则出现，指出保存由 BIOS 提供的内存分布的缓冲区的地址和长度.
 * mmap_addr 是缓冲区的地址,mmap_length 是缓冲区的总大小.
 */
uint32_t mmap_length;
uint32_t mmap_addr;

/* 如果 flags[7] 被置位则出现，则 drives_*域是有效的，
 * 指出第一个驱动器结构的物理地址和这个结构的大小.
 * drives_addr 是地址,drives_length 是驱动器结构的总大小.
 */
uint32_t drives_length;
uint32_t drives_addr;

/* 如果 flags[8] 被置位则出现，则 config_table 域有效，
 * 指出由 GET CONFIGURATION BIOS 调用返回的 ROM 配置表的物理地址.
 * 如果这个 BIOS 调用失败了，则这个表的大小必须是 0.
 */
uint32_t config_table;

/* 如果 flags[9] 则 boot_loader_name 域有效，
 * 包含了引导程序名字在物理内存中的地址.
 * 引导程序名字是正常的 C 风格的以 0 中止的字符串
 */
uint32_t boot_loader_name;
uint32_t apm_table;
```

```
  /* 如果 flags[11] 被置位则 apm_table 域有效,
  包含了如下 APM(高级电源管理) 表的物理地址: */
  uint32_t vbe_control_info;
  uint32_t vbe_mode_info;
  uint32_t vbe_mode;
  uint32_t vbe_interface_seg;
  uint32_t vbe_interface_off;
  uint32_t vbe_interface_len;
} __attribute__((packed)) multiboot_t;
```

在 flags[6] 被置位时候, 则 mmap_*域有效, 指出保存由 BIOS 提供的内存分布的缓冲区的地址和长度, 缓冲区的结构为:

```
/* size: 是相关结构的大小, 单位是字节, 它可能大于最小值 20.
 * base_addr_low: 是启动地址的低 32 位,
 * base_addr_high: 是高 32 位, 启动地址总共有 64 位.
 * length_low: length_low 是内存区域大小的低 32 位.
 * length_high: 是内存区域大小的高 32 位, 总共是 64 位.
 * type: 是内存可用信息 1 代表可用, 其他的代表保留区域
 */

typedef struct mmap_entry_t {
  uint32_t size;
  uint32_t base_addr_low;
  uint32_t base_addr_high;
  uint32_t length_low;
  uint32_t length_high;
  uint32_t type;
} __attribute__((packed)) mmap_entry_t;
```

1.3 GRUB的安装与使用

此处GRUB安装指的是如何将GRUB程序安装到启动硬盘。首先需要在Linux中安装GRUB的一些工具，本次测试使用的版本是_GRUB 2.04，用到的GRUB的相关工具为`grub-install`。

1.3.1 向硬盘中安装GRUB

硬盘的分区:

```
sundong@ubuntu:~/Downloads/minios_grub$ fdisk -l b.img
Disk b.img: 48.85 MiB, 51200000 bytes, 100000 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00000000

Device     Boot Start   End Sectors   Size Id Type
b.img1     *     2048  4000    1953 976.5K 83 Linux
b.img2           4096 99999   95904  46.8M  5 Extended
b.img5           6144 50000   43857  21.4M 83 Linux
b.img6          53248 99999   46752  22.8M 83 Linux
```

其中分区号为1的分区为启动分区（kernel等文件保存在该分区中，os boot引导程序已经写入到该分区的0号扇区）。GRUB的一些可执行代码以及功能模块将被安装在分区号为6的分区。

GRUB代码模块的相关解释如下：GRUB的执行代码保存在boot.img和core.img中，运行时会GRUB的加载程序会将这两个文件分别加载到内存中去，但是这些代码文件都完成的是一些核心功能，GRUB将其他功能分成一个模块（比如ext2.mod对应的就是ext2文件系统的相关操作），这些模块在用到时会被加载进内存。建议不要将grub安装在引导分区。

安装之前的准备操作

- 将b.img文件转化为Linux可识别的设备。`sudo losetup -P /dev/loop15 b.img` 其中具体挂载到哪个回环设备上可以使用`losetup -f`命令查看。
- 将`/dev/loop15p6`挂载到iso目录。`sudo mount /dev/loop15p6 iso`。此处需要提前给`/dev/loop15p6`格式化好文件系统。

安装命令：

```
grub-install --boot-directory=./iso  --modules="part_msdos"  /dev/loop15
```

- `--boot-directory=./iso` 指的是将GRUB的代码和模块文件安装在哪里。此处iso文件挂载的文件系统所在的分区为`/dev/loop15p6`。
- `--modules="part_msdos"` 指的是使用 MS-DOS (MBR)分区格式和分区表。
- `/dev/loop15` 指的是GRUB安装的硬盘（会重写该硬盘的MBR）。

  安装完成之后就可以向硬盘的启动分区写入os boot以及将loader.bin和kernel文件放入到启动分区中。

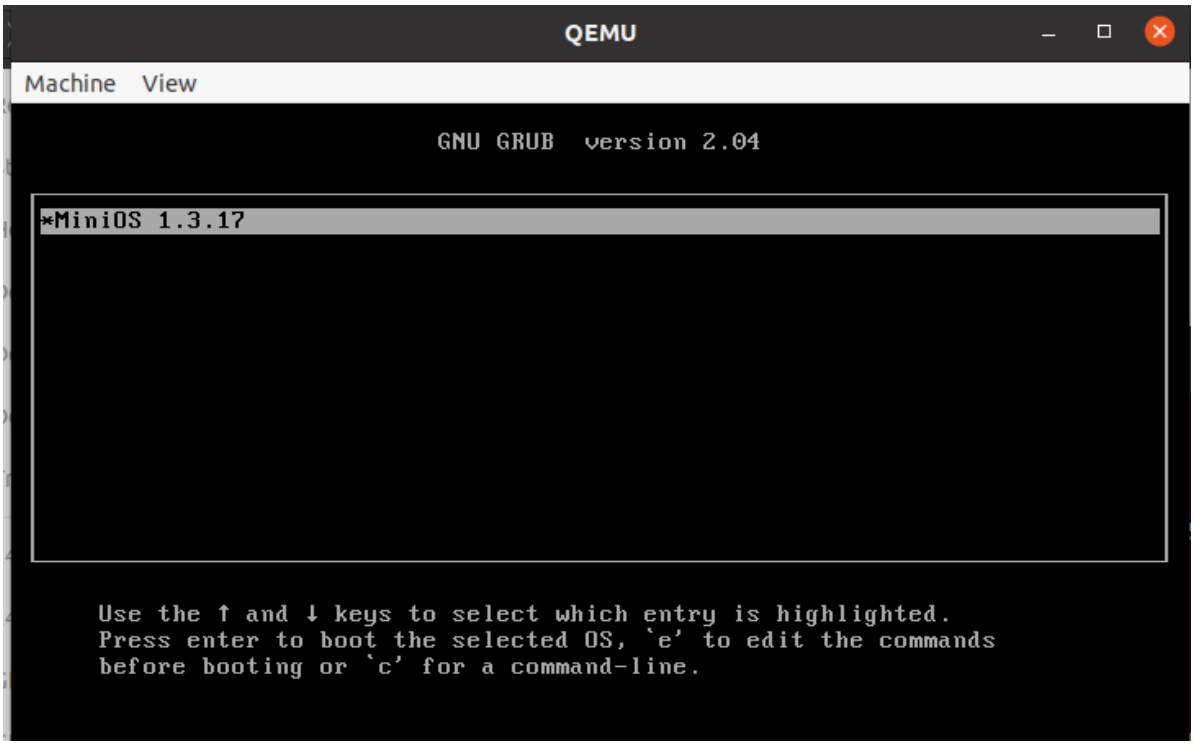
1.3.2 GRUB的交互界面——命令行界面

QEMU启动之后进入GRUB的操作界面如图所示，图中三条指令的含义为：

- `set root=(hd0,msdos1)`,设置启动分区为0号硬盘1号分区，否则默认为grub安装所在的分区。
- `chainloader +1`，加载启动分区中的os boot进内存。
- `boot`，将控制权转交给os boot程序。

之后将执行os 的boot程序。

1.3.3 GRUB的图形操作界面



要想使用GRUB图形界面，只需将grub.cfg这一配置文件放入到grub的安装目录中即可。

grub.cfg文件的内容为：

```
menuentry 'MiniOS 1.3.17'{
set root=(hd0,msdos1)
chainloader +1
boot
}
```

其实就是将一些命令按顺序写入。但是grub.cfg能完成的功能远不止这些，具体用法可以参考[GRUB Simple-configuration](GRUB Simple-configuration).

二、MiniOS启动过程简介

MiniOS的启动过程为标准的链式启动流程，具体如下：

`BIOS加载MBR(共512B)到0x7c00处然后执行MBR --> MBR读取分区表然后将活动分区的Boot(也是512B，存储在活动分区的0号扇区)加载进内存然后执行它 -->Boot在活动分区中寻找loader程序(二进制文件格式，存储的是直接执行的代码)然后加载loader程序到内存然后执行它 --> loader加载kernel程序(elf文件格式)到内存然后正式进入OS`。

2.1 MBR

MBR位于硬盘的0号物理扇区，共512B。前446B存储的是可执行代码，之后的64B存储的是分区表（共4个表项，每个表项16B），之后两个字节存储的是MBR识别标识，固定为0xaa55 。MBR完成的就是分析分区表，识别活动分区，然后将活动分区的第0号扇区（os boot）加载到*0x1000:0x7c00*处，然后执行它。MBR需要向OS Boot程序传递OS Boot所在的分区的起始物理扇区号（比如 2048)。

2.2 Boot

Boot程序位于引导分区的0号扇区（并非硬盘的0号物理扇区）。引导分区的0号扇区的前90字节是FAT32的BPB区,之后的440B的区域为Boot代码存放的区域。Boot中完成的是从启动分区（FAT32文件系统）中搜索loader.bin文件然后，加载到0x9000:100处，然后执行loader.bin，搜索过程中使用BIOS中断int 13扩展功能去读硬盘中的扇区，此时会用到MBR传来的分区的起始扇区的值。

2.3 Loader.bin

loader完成的功能相对较多，它除了在启动分区文件系统中寻找kernel（ELF文件格式）之外，还要完成实模式到保护模式的转换、启动分页机制（双映射）、检查可用的内存信息、解析ELF文件等操作。

注意:

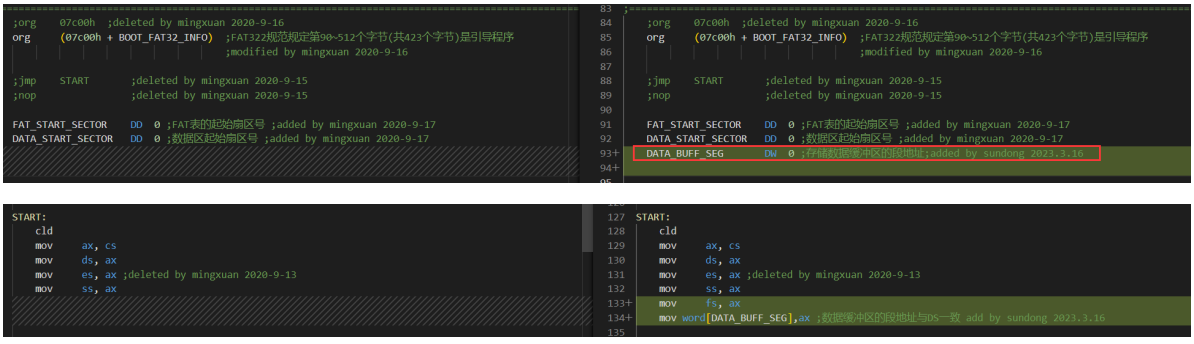
- loader在从文件系统中搜索kernel的过程中使用到的FAT32 BPB数据的位置与Boot所使用的BPB的位置相同（loader和boot共用同一个BPB）位于*0x1000:7c00*处,loader中使用到的起始扇区的这个记录在内存中的位置也与boot所使用的位置相同。
- loader 启动分页机制使用的是双映射，即将0-8M物理地址映射到0-8M线地址的同时也映射到3G--3G+8M这段线地址。原因是loader的代码运行在低地址处，启用页表之后若没映射低地址，loader剩余的代码将无法继续执行。
- loader检查可用的内存段并写入到*0x007ff000* 地址处，然后kernel在初始化budd系统时会读取该内容。

三、MiniOS使用GRUB chainloader引导时需要修改的代码

MiniOS使用GRUB chainloader引导后boot所在的位置变了，不使用GRUB时加载boot在内存的位置是*0x1000:7c00*,使用GRUB引导后boot所在的内存位置为*0x0000:7c00*, 段内偏移没有改变因此代码不需要很大的改动，但是由于段基址发生改变，所以一些数据读写操作会受到影响。不使用GRUB时MBR程序会将引导扇区的起始扇区号放到内存中的一个地址中，boot会去该地址读出起始扇区号，使用GRUB引导后，进入boot时*DS:SI*存储的地址就是启动分区的分区表表项，因此需要boot通过该地址中解析出起始扇区号。本次测试使用的版本为1.3.17.

3.1 boot.asm所作的修改

1. **增加了用于存储数据缓冲区段基址的变量**



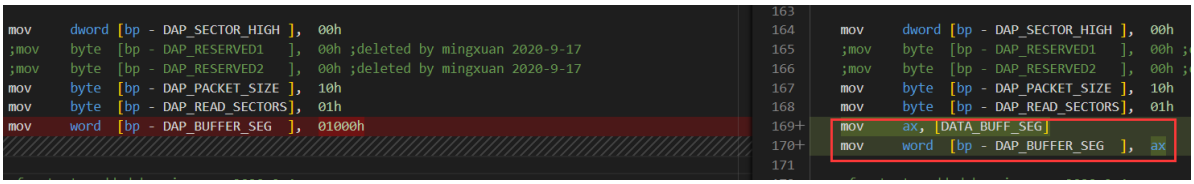
boot在FAT32文件系统中搜索loader.bin时需要将FAT表、目录所在的扇区读入内存，然后遍历。读扇区使用的是BIOS int13 的扩展功能，需要指定读入的地址的段地址和段内偏移，原先版本的实现中这个段地址是采用硬编码的方式写入到代码段的，具体内容为*0x1000*，与数据段一致，更改后将段地址存储到变量中，用到时去变量中取内容。

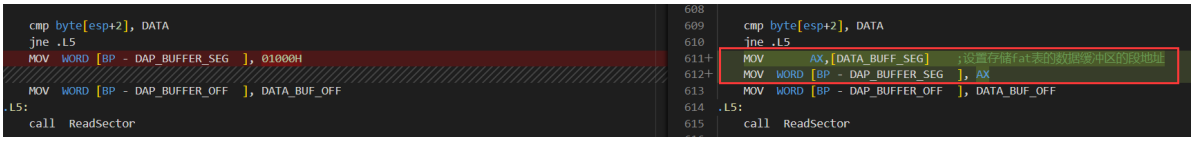
2. **增加了通过grub chainloader传来的分区表表项地址来获取启动分区起始扇区的代码**

```
+         ;grub 链式引导，将当前分区表表项的起始地址放在DS：SI处
+         add     si,OffsetOfStartSecInPET                ;si加上一个起始扇区在表项中的偏移量就是用于存储起始扇区的那
  段内存
+         mov             eax, [ds:si]
+         mov             [OffsetOfActiPartStartSec],eax  ;将起始扇区的物理扇区号放入OffsetOfActiPartStartSec地址
  处 loader中会用到该值
```

*DS:SI*存储的地址为启动分区分区表项，*DS:SI+8*指向的内存存储的就是起始扇区号，改动后将起始扇区号取出并存储到`OffsetOfActiPartStartSec`（*7e00*）处,loader会继续使用该地址获取起始扇区号。

3. **修改了使用BIOS int 13扩展读功能参数初始化的代码**





原来版本中在读扇区到缓冲区时，数据段的初始化代码为硬编码，修改后改为变量中取值，该变量的值在一开始处被初始化为与数据段一致。

四、注意事项

1. MiniOS 1.3.19开始支持了grub引导，具体的说支持**grub的chainloader（链式引导）**。
2. grub chainloader中grub向os_boot传递启动分区分区表项的内存地址，请注意是分区表表项。**主分区表表项和扩展分区中的逻辑分区表项的数据结构是一样的但是分区表项的`起始扇区LBA`这一项存储的值的含义是不同的。主分区分区表项的`起始扇区LBA`就是该分区的起始扇区的物理扇区号，而扩展分区中的逻辑分区表的`起始扇区LBA`存储的是一个偏移量，该偏移量是基于上个分区的起始扇区的。也就是说仅凭扩展分区中逻辑分区的分区表的一个表项是无法确认逻辑分区的起始物理扇区号的，因此MiniOS 暂时不支持在扩展分区的逻辑分区上启动，仅允许在主分区上启动。**
3. **MiniOS 1.3.19对镜像的分区格式发生了调整，镜像改为了100MB，分区也进行了重新划分，重新划分后的分区为：**

```
Device         Boot  Start    End Sectors Size Id Type
hd/test2.img1  *      2048   4095    2048   1M 83 Linux
hd/test2.img2         4096 106495  102400  50M 83 Linux
hd/test2.img3       106496 204799   98304  48M  5 Extended
hd/test2.img5       108544 204799   96256  47M 83 Linux
```

其中分区1格式化为FAT32文件系统，分区2格式化为orangefs文件系统（根文件系统），分区3为扩展分区，分区5为分区3（扩展分区）的逻辑分区。

五、使用grub引导时需要手动修改的地方

Makefile中有两个变量`USING_GRUB_CHAINLOADER`和`BOOT_PART_FS_TYPE`。

```
#added by sundong 2023.3.21
#用于区分是使用grub chainloader
#可选值为 true  false
USING_GRUB_CHAINLOADER = false
#选择启动分区的文件系统格式，目前仅支持fat32和orangfs
BOOT_PART_FS_TYPE= fat32
```

`USING_GRUB_CHAINLOADER` 用于表示是否使用grub引导，若使用grub引导其值应设置为true，除此之外还需要在Linux上安装 `grub-install` 应用。

`BOOT_PART_FS_TYPE` 用于表示启动分区的文件系统类型，目前仅支持orangefs和fat32。

当启动分区使用fat32文件系统时，需要将硬盘启动分区设置为分区1，os/boot/mbr/grub/grub.cfg的配置也需要做修改，具体的设置如下：

硬盘启动分区的设置：

```
Device        Boot  Start    End Sectors Size Id Type
hd/test2.img1 *      2048   4095    2048   1M 83 Linux
hd/test2.img2        4096 106495  102400  50M 83 Linux
hd/test2.img3      106496 204799   98304  48M  5 Extended
hd/test2.img5      108544 204799   96256  47M 83 Linux
```

os/boot/mbr/grub/grub.cfg的配置文件的修改：

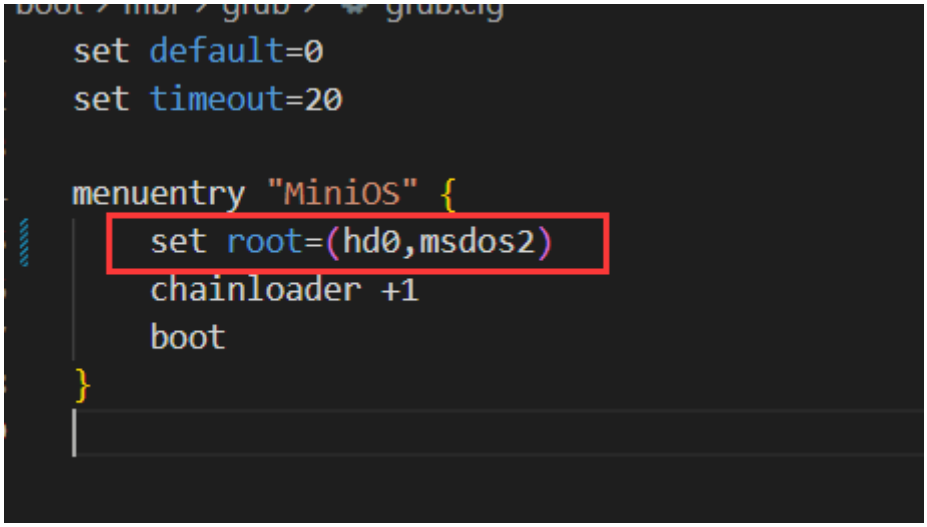
```
set default=0
set timeout=20

menuentry "MiniOS" {
    set root=(hd0,msdos1)
    chainloader +1
    boot
}
```

当启动分区使用OrangeFS文件系统时，需要将硬盘启动分区设置为分区2，os/boot/mbr/grub/grub.cfg的配置需要做修改，具体的设置如下：

硬盘启动分区的设置：

```
Device        Boot  Start    End Sectors Size Id Type
hd/test2.img1       2048   4095    2048   1M 83 Linux
hd/test2.img2 *     4096 106495  102400  50M 83 Linux
hd/test2.img3      106496 204799   98304  48M  5 Extended
hd/test2.img5      108544 204799   96256  47M 83 Linux
```

os/boot/mbr/grub/grub.cfg的配置文件的修改：


set default=0
set timeout=20
menuentry "MiniOS" {
set root=(hd0,msdos2)
chainloader +1
boot
}