

多重引导规范(Multiboot Specification)

翻译:zyj001et (ncs dot mail.ustc.edu.cn)

翻译版本历史

版本记录	V 0.2
修改日期	2005-10-07
修改人	zyj001et(ncs dot mail.ustc.edu.cn)
修改内容	对照原文修改了 V0.1 版本中的不正确的地方

版本记录	V 0.1
创建日期	2005-6-6
创建人	zyj001et(ncs dot mail.ustc.edu.cn)

注意：该版本历史是本翻译文档的版本历史，而不是原文档。

版权信息

本翻译文档遵循 GNU/GPL，你可以自由的修改发布它

1 介绍

这一章粗略地介绍一下 Multiboot 地相关信息，注意，这不是规范本身的一部分。

1.1 Multiboot 规范的背景

每一个操作系统一般都有自己的 Boot Loader。在一个机器上安装一个新的操作系统通常要安装一些全新的启动机制，而这些机制在安装时候和启动时候的用户接口都不同。如果使用传统的 Chaining 机制来使多个操作系统在一台机器上友好相处，那么这是一件很痛苦的事情。对于一个特定的操作系统来说，Boot Loader 方面我们通常是没有选择的余地的。如果操作系统的 Boot Loader 不像你所希望的哪样，或者在你的机器这个 Boot Loader 就不能正常运行，你就要很郁闷了。

对于已经存在的商业操作系统，我们不能解决这个问题，然而对于自由操作系统社区来说，这不是很困难的事情，只需要大家碰到商量一下就可以了。这正是这份规范的目的。简单的来说，它定义了 Boot Loader 和操作系统的接口，任何遵循这个规范的 Boot Loader 可以引导任何一个遵循这个规范的操作系统。这个规范不涉及 Boot Loader 是如何工作等方面的内容，而只是定义了它们如何与要引导的操作系统进行交互。

1.2 适用的体系结构

这份规范主要是用在 PC 上，因为 PC 是目前最常见的架构，而且在 PC 上具有很多不同的操作系统和不同的 Boot Loader。但是，其他的体系架构可能也需要一个引导规范，如果在这个架构上还没有这样的规范的话，这份规范的很多方面，除去 X-86 架构特定的细节，是可以适用到其他的架构上的。

1.3 适用的操作系统

规范适用的操作系统是自由的 32 位操作系统，它必须要能经过简单的修改就能支持这份规范，而不是需要大的改动。这份规范最初是为 Linux、FreeBSD、NetBSD、Mach 以及 VSTa 设计的。我们希望新开发的自由操作系统从一开始就支持这份规范，从而可以利用已经存在的 Boot Loader。当然如果商业操作系统支持这份规范也是非常好的一件事情，然而这很可能是一个梦想。

1.4 启动的介质

我们可以写一个可以从很多介质如软盘、硬盘、网络引导 OS Image 的 Boot Loader。

基于硬盘的 Boot Loader 可能使用很多技术在硬盘上找到相应的 OS Image 以及要启动的 Boot Module，比如通过解释特定的文件系统（如 BSD/Mach 的 Boot Loader），使用预先定义的块列表（如 LILO），或者从一个特殊的 Boot 分区进行引导（如 OS/2）甚至从其他操作系统中引导（如 VSTa 的引导代码是从 DOS 中引导操作系统的）。同样的，基于网络的引导的 Boot Loader 可以使用很多不同的硬件以及协议。

我们希望 Boot Loader 应该支持多重引导机制来增强他们的灵活性、健壮性以及易使用性。

1.5 在启动的时候配置操作系统

通常我们需要在操作系统引导的时候给它一些配置信息。不过这份文档不涉及 Boot Loader 是如何得到这些配置信息的，而只是给 Boot Loader 提供了一个标准的方法来把这些配置信息传递给操作系统。

1.6 怎么使得开发操作系统变得更容易

我们应该能够很容易地创建 OS Image。理想地情况是 OS Image 可以是操作系统平台使用的任何一种 32 位的可执行文件。我们**应该**可以使用 nm 或者 disassembler 来像反汇编一个普通可执行文件一样来反汇编 OS Image，我们不需要使用特定的工具来创建特定的 OS Image。如果这意味需要在 Boot Loader 中多做一些事情，我们也认为是值得的，因为在 Boot Loader 中所占用的内存存在操作系统启动以后是可以被再次使用的，而操作系统所使用的内存是需要一直占用的。操作系统**不应该**需要解决如何进入 32 位模式来 load 在 1M 内存以上的操作系统数据问题，否则这会使得创建 OS Image 更加的困难。

不幸的是，可执行的文件格式非常多，即使是在类 Unix 的 PC 操作系统中也是很多的，而这些格式在每个操作系统中都是不一样的。绝大部分自由操作系统使用 a.out 格式的变体，不过他们中间的一些正在向 ELF 格式迁移。当然，Boot Loader 不需要能处理所有的可执行文件来引导不同格式的 OS Image。否则，Boot Loader 就变成了一个特定的操作系统了。

这份规范在这个问题上做了一个折中，遵循多重引导规范的 OS Image 通常含有一个多重引导头【见 3.1 节】，这个头使得 Boot Loader 可以引导操作系统而不需要知道操作系统使用的 Image 文件是 a.out 格式还是其他格式。这个头不需要放在 OS Image 的一开始。因此，操作系统既可以保持原来的文件格式又可以与多重引导兼容。

1.7 启动模块(Boot Modules)

现代的内核如 VSTa 和 Mach，并不是在内核中包含了所有的功能。它们启动的时候需要其他的软件模块来完成访问设备，挂载文件系统等工作。当然这些模块也可以和内核一起被嵌入到 OS Image 中。但是如果 Boot Loader 能够一开始就独立的引导这些模块的话，对于操作系统和用户来说，会显得更灵活、空间效率更高以及更方便。

因此，这份规范应该给 Boot Loader 提供一个标准的方法，使得 Boot Loader 可以告诉操作系统哪些模块已经被引导了，以及在哪里可以找到这些模块。Boot Loader 不是一定要支持模块引导，但是我们强烈建议这样做。因为如果没有了这个功能，一些操作系统就不能引导了。

2 文档中使用的术语

必须(must) :

当描述 Boot Loader 和 OS Image **一定需要**遵循的规则时候，我们使用术语**必须**。如果他们不遵循这个规则，就不能称为兼容多重引导的。

应该(should) :

当我们描述 Boot Loader 和 OS Image 推荐遵循的规则的时候，我们使用术语**应该**。当然，他们也可以不遵从这些规则。

可以(may) :

当我们在描述 Boot Loader 和 OS Image 被允许遵循的一个规则的时候，我们使用术语**可以**。

Boot Loader :

引导最终运行在机器上的操作系统的任何一个或者一系列的程序。Boot Loader 自身可以包含许多阶段,但是这是具体的实现细节而不是本规范的一部分。只有**最后**一个阶段,即将控制权从 Boot Loader 交给操作系统这一个阶段**必须要**遵循本规范中内容。只有这样,它才是兼容多重引导的。

OS Image :

被 Boot Loader 引导到内存并且将控制权交于的第一个二进制印象文件。通常 OS Image 一个包括内核的可执行文件。

引导模块 :

和内核一起被 Boot Loader 引导进入内存的其他辅助文件,当他们被调用的时候,除了将他们的地址传递给操作系统以外,其他什么事情也不做。

兼容多重引导 :

一个遵循这份文档中一些**必须**的规范的 Boot Loader 或者 OS Image 可以被称为兼容多重引导的。这份文档中的**应该**和**可以**的规则, Boot Loader 或者 OS Image 可以不遵从。

U8 :

8 位无符号数据。

U16 :

16 位无符号数据。由于适用的目标架构是 little-endian¹, 因此 U16 是 little-endian。

U32 :

32 位无符号数据。由于适用的目标架构是 little-endian, 因此 U32 是 little-endian。

U64 :

64 位无符号数据。由于适用的目标架构是 little-endian, 因此 U64 是 little-endian。

3 多重引导规范

规范中,主要阐述了 Boot Loader/OS Image 三个方面内容:

1. Boot Loader 看到的 OS Image 的格式。
2. 当 Boot Loader 启动一个操作系统时候的机器状态。
3. Boot Loader 传递给操作系统的信息格式。

3.1 OS Image 的格式

一个 OS Image 的格式可以是特定操作系统中的普通 32 位可执行文件的标准格式,当然要去掉这些可以被链接到非默认的引导地址如 PC 的 I/O 区域之上或者其他保留地址的文件。当然,OS Image 是不能使用共享库以及其他的特性的。

一个 OS Image 除了含有表示它的格式的头之外,还必须含有一个额外的头,这个头叫做多重引导头[Multiboot header]。多重引导头**必须**在 OS Image 的第一个 8192 字节之内,它还**必须**是长字(32 位)对齐的。通常说来,它**应该**越早出现越好,它还可以被嵌入到正文段的开始,即真正的可执行文件的头的后面²。

¹ little-endian 指数据的对齐方式。如 0X1234 存放在 0X00000000 开始的内存中,那么在 0X00000000 中存放的是 0X34,而 0X00000001 存放的是 0X12。即高位高地址,低位地址。这个是 X86 架构采取的方式。

² 比如 OS Image 的可执行文件格式是 ELF,那么多重引导头可以嵌入到 ELF 头的后面,正文段的开头

3.1.1 多重引导头的布局

多重引导头的布局必须如下所示

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

字段 'magic' 'flags' 'checksum' 在 [3.1.2](#) 节中定义，字段 'header_addr' 'load_addr' 'load_end_addr' 'bss_end_addr' 以及 'entry_addr' 在 [3.1.3](#) 节中定义。字段 'mode_type' 'width' 'height' 'depth' 在 [3.1.4](#) 节中定义。

3.1.2 多重引导头的magic字段³

'magic':

'magic' 是表示多重引导头的魔数。它的值必须是 0X1BADB002⁴。

'flags'

'flags' 表示了一个操作系统需要或者期望 Boot Loader 所具有的一些特性。0-15 位表示需要的特性。如果 Boot Loader 看到这些位中的一位被置，但是因为一些原因它不知道这些位的意思，或者因为一些原因它不能满足这个位所表示的特性这个时候它必须通知用户并且不引导 OS Image。16-31 是可选的特性，如果这些位中一位被设置但是 Boot Loader 不知道这些位的作用，Boot Loader 可以简单地忽略然后正常引导。因此，OS Image 中，没有定义的位的默认值应该 0。这个时候，'flags' 字段用做版本信息和简单的特性选择⁵。

如果 'flags' 中的第 0 位被设置，那么所有和操作系统一起被引导的引导模块必须在页边界(4K)对齐。有的操作系统需要在启动的时候能将包含启动模块的页直接映射到分过页的地址空间(paged space)，因此，它们需要启动模块是页对齐的(page-aligned)。

如果 'flags' 中的第 1 位被设置，那么多重引导信息结构中(见 3.3 节引导信息结构)表示内

³ 注意，这里的 magic 字段包括头中的 'magic' 和 'flags' 和 'checksum' 字段

⁴ 注意，0X1BADB002 是小端存放的

⁵ 可能是不同的定义版本 位的设置不同，因此可以 用来指示 Multiboot 的版本

存容量的信息应该被包含。这表示，结构中至少要含有'mem *'字段。如果 Boot Loader 可以传递一个 memory map('mmap *'字段)，那么它也可以被包含。

如果'flags'中的第 2 位被设置，那么关于 video mode table(见 3.3 节引导信息结构)必须对内核可见。

如果'flags'中的第 16 位被设置，那么多重引导头中的偏移 8-24⁶中的字段是有效的。Boot Loader**应该**使用他们而不是真正可执行文件的头来计算到哪里去引导 OS Image。如果内核是 ELF 格式的，这些信息**不是**必须要提供的，但是如果是 a.out 或者其他格式的，那么就**必须**提供。兼容多重引导的 Boot Loader**必须**可以引导 ELF 格式的印象文件或者是在多重引导头中包含引导地址的其他格式文件。当然，它们也可以支持其他的可执行文件，比如很多的 a.out 变体，但是**不是**必须要的。

Checksum

字段 checksum 是 32 位的无符号数，它加上其他的 magic 字段（如 magic 和 flags）的和必须是 32 位的无符号的 0。

3.1.3 多重引导头的地址字段

当 flag 字段的第 16 位被设置时，地址字段才有效。所有的地址字段中的地址都是物理地址，这些字段的含义如下：

head_addr

包含多重引导头的开始地址 - 在那里去寻找 magic 值的**物理地址**。这个字段用来同步 OS Image 偏移和物理地址之间的映射。

Load_addr

包含正文段的开始物理地址。OS Image 中偏移（表示从哪里开始引导）是由多重引导头的偏移（表示在哪里能找到多重引导头）定义的，它们的关系是减（header_addr-load_addr）⁷，load_addr**必须**小于等于 head_addr。

Load_end_addr

包含数据段结束的物理地址。（Load_end_addr-load_addr）表示要引导多少数据。这表示在 OS Image 里面正文段和数据段**必须**是连续的。这对于 a.out 格式的文件来说符合的。如果这个值是 0，boot loader 认为数据和正文段占据了整个 OS Image。

Bss_end_addr

包含 bss 段结束的物理地址。Boot Loader 将这片区域初始化为 0，并且保留这片内存以避免被引导模块以及操作系统中的其他数据占据。如果这个字段为 0，boot loader 认为目前没有 bss 段。

Entry_addr

Boot Loader 应该 jump 到并且开始执行操作系统的地址。

3.1.4 多重引导头的图形字段

如果 flag 字段的第二位被设置，那么所有的图形字段有效。他们表示了图形模式。注意这些数据仅仅是 OS Image 的**推荐模式**(recommended mode)。如果模式存在而用户又没有

⁶ 应该是 12-28 吧，文档错误还是我理解错误？

⁷ 这一段的翻译还需要斟酌

显式地指定一个模式的话，Boot Loader **应该**设置这些模式。否则如果简单模式可见的话，Boot Loader **应该**回到简单模式。

Mode_type

0 表示线形图像模式，1 表示 EGA-标准文字模式。其他的值由以后扩展使用。注意即使是这个字段值为 0，Boot Loader **可以**设置为文字模式。

Width

包含列数。在图形模式下表示像素，在文字模式下表示字符。0 表示 OS Image 没有指定。

Height

包含行数。在图形模式下表示像素，在文字模式下表示字符。0 表示 OS Image 没有指定。

Dept

在图形模式下表示一个像素点的 bit 数目，在文字模式下必须为 0。0 表示 OS Image 没有指定。

3.2 机器状态

当 boot loader 引导 32 位操作系统的时候，机器**必须**有如下的状态：

EAX：

必须包含魔数 0x2BADB002 这个值告诉操作系统目前它是由兼容的 Multiboot 的 boot loader 引导的。

EBX：

必须包含 boot loader 提供的多重引导信息结构(见 3.3 节多重信息引导结构)的 32 位物理地址。

CS：

必须是 32 位的读/执行的代码段，偏移是 0 以及界限是 0xFFFFFFFF。具体值没有定义。

DS

ES

FS

GS

SS：

必须是 32 位的读/执行数据段，偏移是 0 以及界限是 0xFFFFFFFF。具体值没有定义。

A20 GATE：

必须 enable。

CR0:

31 位(PG)**必须**清除，第 0 位(PE)**必须**设置。其他位没有定义。

EFLAGS：

第 17(VM)位**必须**清除，第 9 位(IF)**必须**清除，其他位没有定义。

处理器寄存器的其他标志位没有定义。特别的，这包括

ESP：

如果需要的话，OS Image 可以包含它自己的堆栈。

GDTR：

即使段寄存器按照上面所说的设置，GDTR 这个时候也可能不合法。因此，在它设置自己的 GDT 之前，不要 load 任何的段寄存器（即使是重新 reload 同样的值）。

IDTR：

OS Image 在设置自己的 IDT 之前**必须**关闭中断。

但是，Boot Loader **应该**保留其他的机器状态在正常的值，即被 BIOS 初始化的值（在 Boot Loader 从 DOS 运行的话，**应该**是被 DOS 初始化的值）应该保持原来的值。也就是说在被引导以后，只要它不覆盖 BIOS 数据结构，操作系统就**应该**可以使用 BIOS 调用。并且，Boot Loader **必须**保持可编程 PIC 的值是正常的 BIOS/DOS 值，即使在切换到 32 位方式的时候 boot loader 改变了他们的值。

3.3 引导信息结构

在进入操作系统前，EBX 寄存器包含多重引导信息结构的物理地址，通过这个结构，Boot Loader 可以和操作系统交换重要的信息。操作系统可以使用或者忽略这个结构中的任何一部分，也就是说从 Boot Loader 传递来的所有信息只是 advisor。

多重引导信息结构以及它的子结构可以被 Boot Loader 放在内存中的任何地方（当然除了位内核和引导模块保留的部分）。在操作系统使用完这部分信息之前，操作系统必须保证存放这个信息的内存不被覆盖。

多重引导信息结构(到目前定义的为止)如下图：

0	flags	(required)
4	mem_lower	(present if flags[0] is set)
8	mem_upper	(present if flags[0] is set)
12	boot_device	(present if flags[1] is set)
16	cmdline	(present if flags[2] is set)
20	mods_count	(present if flags[3] is set)
24	mods_addr	(present if flags[3] is set)
28 - 40	syms	(present if flags[4] or flags[5] is set)
44	mmap_length	(present if flags[6] is set)
48	mmap_addr	(present if flags[6] is set)
52	drives_length	(present if flags[7] is set)
56	drives_addr	(present if flags[7] is set)
60	config_table	(present if flags[8] is set)
64	boot_loader_name	(present if flags[9] is set)

68	apm_table		(present if flags[10] is set)
72	vbe_control_info		(present if flags[11] is set)
76	vbe_mode_info		
80	vbe_mode		
82	vbe_interface_seg		
84	vbe_interface_off		
86	vbe_interface_len		

第一个 longword 表示多重引导信息结构中其他字段的正确性。到目前为止没有被定义的位**必须**被 Boot Loader 设置成 0。任何一个被设置为 1 但是操作系统不能解析的位**应该**被忽略。因此，'flags' 字段也表示了版本的信息，它使得多重引导信息结构可以在将来被扩充而不用破坏已经有的定义。

如果 'flags' 中的第 0 位被设置，那么表示 'mem *' 字段有效。'mem_lower' 以及 'mem_upper' 分别表示 lower 和 upper 内存的数量，单位是 KB。Lower 内存开始于地址 0，upper 内存开始于地址 1M。lower 内存的最大可能值是 640KB。Upper 内存返回的值是第一个 upper 内存洞的最大地址减去 1M 后的值(但是不保证是这个值)。

如果 'flags' 中的第 1 位被设置，那么 'boot_device' 字段是有效的，表示 boot loader 从一个 BIOS 磁盘设备引导 OS Image。如果 OS Image 没有被从 BIOS 磁盘引导，'boot_device' 字段**必须**不存在(第 3⁸位**必须**被清除)。操作系统可以使用这个字段作为确定自己 root 设备的线索(不是一定要这样做)。「boot_device」字段存在与 4 个单字节的子字段中。

drive	part1	part2	part3
-------	-------	-------	-------

第一个字节表示 BIOS INT 13 能理解的 BIOS 驱动器号，比如 0X00 表示第一个软盘或者 0X80 表示第一个硬盘。

接下来的三个字节表示了 boot 分区。「part 1」表示了最高级的分区号，「part 2」表示了高级分区的子分区。分区号通常从 0 开始。没有使用的分区字节**必须**设置成 0XFF。例如，如果磁盘使用一级 DOS 分区模式分区，那么「part 1」含有 DOS 分区号，「part 2」「part 3」都是 0XFF。如果磁盘首先使用 DOS 分区模式分区，然后其中的一个 DOS 分区又使用 BSD 的 disklabel 策略进行分区，分成几个 BSD 的分区，那么「part 1」包含 DOS 分区号，「part 2」在 DOS 分区中的 BSD 子分区号，「part 3」设置成 0XFF。

DOS 扩展分区的分区号从 4 开始，而不是作为嵌套(nested) 的子分区，即使扩展分区在磁盘的 layout 上通常是金字塔结构的。例如，Boot Loader 从一个常见的 DOS 风格分区的盘上的第二个扩展分区引导操作系统的时候，「part 1」是 5，「part 2」和「part 3」都是 0XFF。

如果 'flags' 字段的第 2 位被设置，'cmdline' 字段有效，并且包含要传递给内核的命令行的物理地址。命令行通常是 C-style 的以 0 结尾的字符串。

如果 'flags' 字段的第 3 位被设置，'mods' 字段表示哪些引导模块将要和内核一起引导，并且在哪儿可以找到它们。「mods_count」包含要引导的引导模块的数目，「mods_addr」包含了第一个引导模块结构 [module structure] 的物理地址。即使 'flags' 字段的第 1⁹ 位被设置，「mods_count」也可以是 0，表示没有引导模块要引导。每一个引导模块结构 [module

⁸ 是第 1 位吧，因为第 1 位是 enable 字段「boot_device」的啊

⁹ 是第 3 位吧，因为第 3 位是 enable 引导模块相关字段的

structure]的格式为

	+-----+
0	mod_start
4	mod_end
	+-----+
8	string
	+-----+
12	reserved (0)
	+-----+

前两个字段表示引导模块本身的开始和结束地址，'string'字段指定了一个与特殊的引导模块相关的字符串。它是以 0 结尾的 ASCII 字符串，和内核命令行一样。如果没有字符串与引导模块相关联，那么'string'字段是 0。通常说来，这个字符串一般是一个命令行（比如操作系统把引导模块看成一个可执行程序）或者一个路径（例如操作系统把引导模块看成一个文件），它的最精确的用途是由操作系统指定的。'reserved'字段**必须**被 Boot Loader 设置成 0，操作系统**必须**忽略它。

注意：第 4 位和第 5 位是互斥的。

如果'flags'字段的第 4 位被设置，那么在多重引导信息结构中第 28 字节开始的字段是有效的：

	+-----+
28	tabsize
32	strsize
36	addr
40	reserved (0)
	+-----+

这表示一个 a.out 内核印象中在哪里可以找到符号表。'addr'表示 a.out 格式中 nlist 结构数组大小(4 字节无符号 long)的物理地址，紧跟在后面的的是数组本身，然后是以 0 结尾的字符串集合的大小(4 字节无符号 long)，最后是字符串集合本身。'tabsize'等于它的大小参数(在符号段开始可以找到)，'strsize'等于接下来的字符串表的大小参数(在字符串段的开始可以找到)，字符串表是供符号表参考的。注意即使'flags'的第 4 位被设置，'tabsize'也可以为 0，表示没有符号。

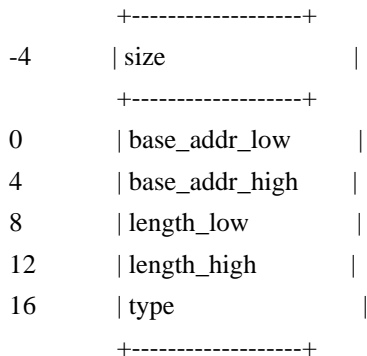
如果'flags'字段的第 5 位被设置，那么在多重引导信息结构中第 28 字节开始的字段是有效的：

	+-----+
28	num
32	size
36	addr
40	shndx
	+-----+

这表示一个 ELF 内核的 section header 表的地址，每一个 entry 的大小，entries 的数目以及作为名字索引的字符串表。他们对应于 ELF 规范中程序头的'shdr_*'（'shdr_num'等等）。所有的 sections 被 load 进来以后，ELF section header 中的物理地址就代表这些 sections 在内存中的位置（可以参考 I386 ELF 文档，里面有如何读取 section header 的描述）。注意，'flags'字段的第 5 位被设置，'shdr_sum'也可能为 0，表示没有符号。

如果'flags'字段的第 6 位被设置，那么'mmap_*'字段有效，表示一个缓冲区的地址和长

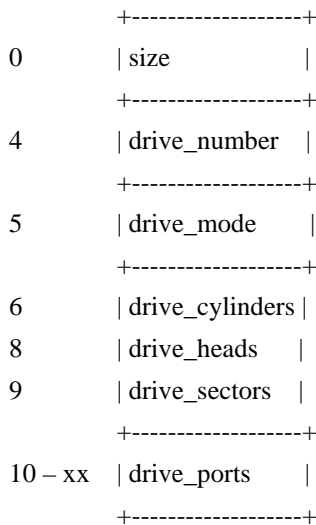
度,这个缓冲区含有由 BIOS 提供的 memory map。'mmap_addr'是地址,'mmap_length'是缓冲区的总长度。缓冲区中含有一个或者多个下面的 size/structure 对。(size 被用来跳到下一个 size/structure 对)



'size'是对应的 structure 的大小(字节为单位),最小是 20 字节。'base_addr_low'是开始地址的低 32 位的,'base_addr_high'是高 32 位地址,这样,开始地址是总共 64 位的。'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,因此,长度总共是 64 位。'type'是地址区域的类型,1 表示可用的 RAM,其他值都表示一个保留的区域。

这个 map 保证列出所有可以用在正常用途的 RAM。

如果'flags'字段的第 7 位被设置,那么字段'drives_*'有效,表示第一个 drive 结构[drive structure]的物理地址以及 drive 结构的大小。'drives_addr'是地址,'drives_length'是所有 drive 结构的总大小。注意,'drives_length'可以是 0。每一个 drive 结构如下图所示:



'size'字段表示这个结构的大小。大小与端口的多少有关。注意,大小可能不等于(10+2*端口数)。这主要是由于对齐(alignment)的原因。

'drive_number'字段含有 BIOS 的驱动器号。'drive_mode'字段表示 boot loader 使用的访问模式。当前,定义了如下的模式义:

0 : CHS 模式。(传统的 柱面/头/扇区 地址模式)

1 : LBA 模式(逻辑块地址模式)

'drive_cylinders' 'drive_heads' 和 'drive_sectors' 表示一些 BIOS 探测到的参数。'drive_cylinders'表示柱面数目,'drive_heads'表示头的数目,'drive_sectors'表示一个轨道上的扇区数目。

'drive_ports'字段包含一个数组,数组的值是 BIOS 中使用的驱动器的 I/O 端口的代码。

这个 array 含有 0 个或者多个无符号的双字节整数，以 0 结尾。注意 array 可能含有不是与真正的驱动器有关的 I/O 端口的数字(如 DMA 控制器的端口)。

如果'flags'字段的第 8 位被设置，那么'config_table'字段有效，表示 BIOS 调用 GET CONFIGURATION 所返回的 ROM 配置表的地址，这个表的大小必须为 0。

如果'flags'字段的第 9 位被设置，'boot_loader_name'字段有效，值为引导内核的 boot loader 名字字符串的地址。名字字符串是以 0 结尾的 C-style 字符串。

如果'flags'字段的第 10 位被设置，'apm_table'字段有效，包含有 APM 表的物理地址。APM 表结构如下：

+-----+		
0	version	
2	cseg	
4	offset	
8	cseg_16	
10	dseg	
12	flags	
14	cseg_len	
16	cseg_16_len	
18	dseg_len	
+-----+		

字段'version' 'cseg' 'offset' 'cseg_16' 'dseg' 'flags' 'cseg_len' 'cseg_16_len' 'dseg_len'表示版本号，32 位保护模式的代码段，entry point的偏移，16 位包含模式的代码段，16 位保护模式的数据段，标志，32 位保护模式的代码段长度，16 位保护模式的代码段长度以及 16 位保护模式数据库长度。只有'offset'是 4 字节，其他都是 2 字节。可以参看 [高级电源管理 \(APM\)BIOS接口规范](#)来获得更多的信息。

如果'flags'字段的第 11 位被设置，图形表[graphics table]有效。只有内核在多重引导头中表示它接受图形模式的情况下，这个字段才能被设置。

字段'vebe_control_info'和'vebe_mode_info'分别含有由 VBE 00H 功能返回的 VBE 控制信息的物理地址和由 VBE 01H 功能返回的 VBE 模式信息。

'vebe_mode'以 VBE 3.0 规范中的格式表明当前 video 模式。

其他的字段'vebe_interface_seg' 'vebe_interface_off' 以及'vebe_interface_len' 包含了在 VBE 2.0+规范中定义的保护模式接口表。如果这个信息不可用的话，这些字段的值被设置为 0。注意 VBE 3.0 定义了它的保护模式接口，这个接口与旧的不兼容。如果你要使用你的保护模式接口。你可能要自己去寻找表。

图形表的字段是为 VBE 设计的，但是 multiboot Boot Loader 可能会在非 VBE 模式下模拟 VBE，这样他们就像 VBE 模式一样。

4 例子

注意：下面的部分不是规范文档的一部分，但是，对于操作系统和 boot loader 开发人员是一个参考。

4.1 PC 上要注意的问题

考参多重引导信息结构中'flags'标志的第 0 位，如果 boot loader 使用了一个旧的 BIOS

或者最新的 BIOS 不可用(参考第 6 位的描述),那么 BIOS 将报告有 15M 或者 63M 的可用内存。我们强烈**建议** Boot Loader 执行一个彻底的内存容量探测。

考参多重引导信息结构中'flags'标志的第 6 位,我们要注意这里使用的数据结构(开始与'baseAddrLow')是由 INT 15H AX=E820(Query System Address Map Call)返回的。你可以查看 GRUB 用户手册中的“Query System Address Map”部分来获得更多信息。这个接口使得 Boot Loader 可以不用修改的使用任何一个 BIOS 接口的有用扩展,可以传递给操作系统任何想要的额外数据。

4.2 BIOS 的设备映射(device mapping)技术

这个技术**应该**可以在不需要在设备驱动器本身任何特别的支持的情况下就可以被任何一个 PC 操作系统使用,本部分将详细讨论具体的细节,特别是 I/O restriction technique 部分。

一般来说,数据比较技术比较快但是它是不彻底的解决方法。在大部分的时候,它可以工作正常,但是它不是每个时候都会,并且它相对来说比较简单。

I/O restriction technique 相比之下更复杂,但是它在任何情况下都能解决问题,而且即使在不是所有的设备都有操作系统的设备驱动程序的情况下,也可以访问 BIOS 设备。

4.2.1 数据比较技术

在激活任何一个设备驱动之前,我们需要从每个磁盘上的相似扇区获得足够的信息,这样我们才能给每个磁盘一个唯一的标识。

在激活设备驱动器后,使用操作系统驱动来比较从驱动器获得的数据。通常,这些数据对于提供映射来说是足够的了。

问题:

1. 一些 BIOS 设备上的数据可能是一样的(所以从 BIOS 上读取驱动器的部分需要有放弃的机制)。
2. 可能有一些驱动器不能被 BIOS 访问,但是这些驱动器和一些 BIOS 使用的驱动器是一样的(所以也应该有放弃的能力)。¹⁰

4.2.2 I/O restriction technique

首先创建一个 copy-on-write 的映射用来让设备驱动器将数据写回 RAM(这一步可能不需要)。然后为将要创建的新的 BIOS 虚拟机保持原始数据的拷贝。

对于每一个在线的设备驱动器,我们可以通过如下的方式来获得哪一个是不能访问的:

1. 创建一个新的 BIOS 虚拟机
2. 为设备驱动器所要求的区域创建一个 I/O permission 映射,对这个映射不能做任何操作(不可读也不可写)
3. 访问每一个设备。
4. 记录访问哪一个设备成功,哪一个设备尝试去访问 restricted I/O 区域(可能需要一个 xor 运算)。

对于每一个设备驱动器,根据它包含了多少个 BIOS 设备的信息(在这个 list 中,不能有

¹⁰ 这段话不是很理解,所以翻译的很生硬。

空隙), 我们可以很容易知道在控制器上有多少个设备。

一般来说, 对于给定的BIOS数字, 在每一个控制器上至少有两个磁盘。但是他们常常从设备控制器上最低的逻辑数字设备开始计数¹¹。

4.3 例子 OS 代码

这一节中, 我们的例子是多重引导的内核“kernel”。内核只是在屏幕上打印多重引导信息结构, 所以你可以使用这个内核去测试一个兼容多重引导的内核, 或者作为实现一个多重引导内核的参考。源代码可以在 GRUB 的目录‘doc’下找到。

内核‘kernel’只包含 3 个文件: ‘boot.S’ ‘kernel.c’ ‘multiboot.h’。其中‘boot.S’中的汇编代码是用 GAS(参考 GNU 汇编中内容)书写的, 它包含规范中描述的多重引导信息结构。当一个兼容多重引导的 Boot Loader 引导它的时候, 它初始化堆栈指针以及 EFLAGS, 然后调用‘kernel.c’中定义的函数 cmain。如果 cmain 返回到 callee, 然后它会打印一条信息告诉用户内核停止的信息, 在你按下重启键之前一直停止。文件‘kernel.c’包含函数 cmian, 它检查 Boot Loader 传递来的魔数是否正确, 并且定义了屏幕上打印信息的一些函数。文件‘multiboot.h’定义了一些宏, 比如多重引导头的魔数, 以及多重引导头和多重引导信息结构。

4.3.1 multiboot.h

```
/* multiboot.h - the header for Multiboot */
```

```
/* Copyright (C) 1999, 2001 Free Software Foundation, Inc.
```

```

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.
```

```

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
```

```

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */
```

```
/* Macros. */
```

```
/* The magic number for the Multiboot header. */
```

```
#define MULTIBOOT_HEADER_MAGIC          0x1BADB002
```

¹¹ 不是很理解这个是什么意思。

```

/* The flags for the Multiboot header. */
#ifdef __ELF__
#define MULTIBOOT_HEADER_FLAGS      0x00000003
#else
#define MULTIBOOT_HEADER_FLAGS      0x00010003
#endif

/* The magic number passed by a Multiboot-compliant boot loader. */
#define MULTIBOOT_BOOTLOADER_MAGIC  0x2BADB002

/* The size of our stack (16KB). */
#define STACK_SIZE                   0x4000

/* C symbol format. HAVE_ASM_USCORE is defined by configure. */
#ifdef HAVE_ASM_USCORE
#define EXT_C(sym)                   _ ## sym
#else
#define EXT_C(sym)                   sym
#endif

#ifndef ASM
/* Do not include here in boot.S. */

/* Types. */

/* The Multiboot header. */
typedef struct multiboot_header
{
    unsigned long magic;
    unsigned long flags;
    unsigned long checksum;
    unsigned long header_addr;
    unsigned long load_addr;
    unsigned long load_end_addr;
    unsigned long bss_end_addr;
    unsigned long entry_addr;
} multiboot_header_t;

/* The symbol table for a.out. */
typedef struct aout_symbol_table
{
    unsigned long tabsize;
    unsigned long strsize;
    unsigned long addr;

```

```

    unsigned long reserved;
} aout_symbol_table_t;

/* The section header table for ELF. */
typedef struct elf_section_header_table
{
    unsigned long num;
    unsigned long size;
    unsigned long addr;
    unsigned long shndx;
} elf_section_header_table_t;

/* The Multiboot information. */
typedef struct multiboot_info
{
    unsigned long flags;
    unsigned long mem_lower;
    unsigned long mem_upper;
    unsigned long boot_device;
    unsigned long cmdline;
    unsigned long mods_count;
    unsigned long mods_addr;
    union
    {
        {
            aout_symbol_table_t aout_sym;
            elf_section_header_table_t elf_sec;
        } u;
    }
    unsigned long mmap_length;
    unsigned long mmap_addr;
} multiboot_info_t;

/* The module structure. */
typedef struct module
{
    unsigned long mod_start;
    unsigned long mod_end;
    unsigned long string;
    unsigned long reserved;
} module_t;

/* The memory map. Be careful that the offset 0 is base_addr_low
    but no size. */
typedef struct memory_map
{

```

```

    unsigned long size;
    unsigned long base_addr_low;
    unsigned long base_addr_high;
    unsigned long length_low;
    unsigned long length_high;
    unsigned long type;
} memory_map_t;

#endif /* ! ASM */

```

4.3.2 boot.S

```
/* boot.S - bootstrap the kernel */
```

```
/* Copyright (C) 1999, 2001 Free Software Foundation, Inc.
```

```

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

```

```

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

```

```

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */

```

```

#define ASM      1
#include <multiboot.h>

    .text

    .globl  start, _start
start:
_start:
    jmp     multiboot_entry

    /* Align 32 bits boundary. */
    .align 4

    /* Multiboot header. */
multiboot_header:

```

```

        /* magic */
        .long  MULTIBOOT_HEADER_MAGIC
        /* flags */
        .long  MULTIBOOT_HEADER_FLAGS
        /* checksum */
        .long  -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
#ifdef __ELF__
        /* header_addr */
        .long  multiboot_header
        /* load_addr */
        .long  _start
        /* load_end_addr */
        .long  _edata
        /* bss_end_addr */
        .long  _end
        /* entry_addr */
        .long  multiboot_entry
#endif /* ! __ELF__ */

```

multiboot_entry:

```

        /* Initialize the stack pointer. */
        movl   $(stack + STACK_SIZE), %esp

        /* Reset EFLAGS. */
        pushl  $0
        popf

        /* Push the pointer to the Multiboot information structure. */
        pushl  %ebx
        /* Push the magic value. */
        pushl  %eax

        /* Now enter the C main function... */
        call   EXT_C(cmain)

        /* Halt. */
        pushl  $halt_message
        call   EXT_C(sprintf)

```

```

loop:   hlt
        jmp   loop

```

```

halt_message:
        .asciz "Halted."

```

```
/* Our stack area. */  
.comm stack, STACK_SIZE
```

4.3.3 kernel.c

```
/* kernel.c - the C part of the kernel */
```

```
/* Copyright (C) 1999 Free Software Foundation, Inc.
```

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */

```
#include <multiboot.h>
```

```
/* Macros. */
```

```
/* Check if the bit BIT in FLAGS is set. */
```

```
#define CHECK_FLAG(flags,bit) ((flags) & (1 << (bit)))
```

```
/* Some screen stuff. */
```

```
/* The number of columns. */
```

```
#define COLUMNS 80
```

```
/* The number of lines. */
```

```
#define LINES 24
```

```
/* The attribute of an character. */
```

```
#define ATTRIBUTE 7
```

```
/* The video memory address. */
```

```
#define VIDEO 0xB8000
```

```
/* Variables. */
```

```
/* Save the X position. */
```

```
static int xpos;
```

```
/* Save the Y position. */
```

```

static int ypos;
/* Point to the video memory. */
static volatile unsigned char *video;

/* Forward declarations. */
void cmain (unsigned long magic, unsigned long addr);
static void cls (void);
static void itoa (char *buf, int base, int d);
static void putchar (int c);
void printf (const char *format, ...);

/* Check if MAGIC is valid and print the Multiboot information structure
   pointed by ADDR. */
void
cmain (unsigned long magic, unsigned long addr)
{
    multiboot_info_t *mbi;

    /* Clear the screen. */
    cls ();

    /* Am I booted by a Multiboot-compliant boot loader? */
    if (magic != MULTIBOOT_BOOTLOADER_MAGIC)
    {
        printf ("Invalid magic number: 0x%x\n", (unsigned) magic);
        return;
    }

    /* Set MBI to the address of the Multiboot information structure. */
    mbi = (multiboot_info_t *) addr;

    /* Print out the flags. */
    printf ("flags = 0x%x\n", (unsigned) mbi->flags);

    /* Are mem_* valid? */
    if (CHECK_FLAG (mbi->flags, 0))
        printf ("mem_lower = %uKB, mem_upper = %uKB\n",
                (unsigned) mbi->mem_lower, (unsigned) mbi->mem_upper);

    /* Is boot_device valid? */
    if (CHECK_FLAG (mbi->flags, 1))
        printf ("boot_device = 0x%x\n", (unsigned) mbi->boot_device);

    /* Is the command line passed? */

```

```

if (CHECK_FLAG (mbi->flags, 2))
    printf ("cmdline = %s\n", (char *) mbi->cmdline);

/* Are mods_* valid? */
if (CHECK_FLAG (mbi->flags, 3))
{
    module_t *mod;
    int i;

    printf ("mods_count = %d, mods_addr = 0x%x\n",
            (int) mbi->mods_count, (int) mbi->mods_addr);
    for (i = 0, mod = (module_t *) mbi->mods_addr;
         i < mbi->mods_count;
         i++, mod += sizeof (module_t))
        printf (" mod_start = 0x%x, mod_end = 0x%x, string = %s\n",
                (unsigned) mod->mod_start,
                (unsigned) mod->mod_end,
                (char *) mod->string);
}

/* Bits 4 and 5 are mutually exclusive! */
if (CHECK_FLAG (mbi->flags, 4) && CHECK_FLAG (mbi->flags, 5))
{
    printf ("Both bits 4 and 5 are set.\n");
    return;
}

/* Is the symbol table of a.out valid? */
if (CHECK_FLAG (mbi->flags, 4))
{
    aout_symbol_table_t *aout_sym = &(mbi->u.aout_sym);

    printf ("aout_symbol_table: tabsize = 0x%0x, "
            "strsize = 0x%x, addr = 0x%x\n",
            (unsigned) aout_sym->tabsize,
            (unsigned) aout_sym->strsize,
            (unsigned) aout_sym->addr);
}

/* Is the section header table of ELF valid? */
if (CHECK_FLAG (mbi->flags, 5))
{
    elf_section_header_table_t *elf_sec = &(mbi->u.elf_sec);

```

```

printf ("elf_sec: num = %u, size = 0x%x,"
        " addr = 0x%x, shndx = 0x%x\n",
        (unsigned) elf_sec->num, (unsigned) elf_sec->size,
        (unsigned) elf_sec->addr, (unsigned) elf_sec->shndx);
}

/* Are mmap_* valid? */
if (CHECK_FLAG (mbi->flags, 6))
{
    memory_map_t *mmap;

    printf ("mmap_addr = 0x%x, mmap_length = 0x%x\n",
            (unsigned) mbi->mmap_addr, (unsigned) mbi->mmap_length);
    for (mmap = (memory_map_t *) mbi->mmap_addr;
         (unsigned long) mmap < mbi->mmap_addr + mbi->mmap_length;
         mmap = (memory_map_t *) ((unsigned long) mmap
                                   + mmap->size + sizeof (mmap->size)))
        printf (" size = 0x%x, base_addr = 0x%x%x,"
                " length = 0x%x%x, type = 0x%x\n",
                (unsigned) mmap->size,
                (unsigned) mmap->base_addr_high,
                (unsigned) mmap->base_addr_low,
                (unsigned) mmap->length_high,
                (unsigned) mmap->length_low,
                (unsigned) mmap->type);
    }
}

/* Clear the screen and initialize VIDEO, XPOS and YPOS. */
static void
cls (void)
{
    int i;

    video = (unsigned char *) VIDEO;

    for (i = 0; i < COLUMNS * LINES * 2; i++)
        *(video + i) = 0;

    xpos = 0;
    ypos = 0;
}

/* Convert the integer D to a string and save the string in BUF. If

```

```

        BASE is equal to 'd', interpret that D is decimal, and if BASE is
        equal to 'x', interpret that D is hexadecimal. */
static void
itoa (char *buf, int base, int d)
{
    char *p = buf;
    char *p1, *p2;
    unsigned long ud = d;
    int divisor = 10;

    /* If %d is specified and D is minus, put '-' in the head. */
    if (base == 'd' && d < 0)
    {
        *p++ = '-';
        buf++;
        ud = -d;
    }
    else if (base == 'x')
        divisor = 16;

    /* Divide UD by DIVISOR until UD == 0. */
    do
    {
        int remainder = ud % divisor;

        *p++ = (remainder < 10) ? remainder + '0' : remainder + 'a' - 10;
    }
    while (ud /= divisor);

    /* Terminate BUF. */
    *p = 0;

    /* Reverse BUF. */
    p1 = buf;
    p2 = p - 1;
    while (p1 < p2)
    {
        char tmp = *p1;
        *p1 = *p2;
        *p2 = tmp;
        p1++;
        p2--;
    }
}

```

```

/* Put the character C on the screen. */
static void
putchar (int c)
{
    if (c == '\n' || c == '\r')
        {
            newline:
            xpos = 0;
            ypos++;
            if (ypos >= LINES)
                ypos = 0;
            return;
        }

    *(video + (xpos + ypos * COLUMNS) * 2) = c & 0xFF;
    *(video + (xpos + ypos * COLUMNS) * 2 + 1) = ATTRIBUTE;

    xpos++;
    if (xpos >= COLUMNS)
        goto newline;
}

/* Format a string and print it on the screen, just like the libc
   function printf. */
void
printf (const char *format, ...)
{
    char **arg = (char **) &format;
    int c;
    char buf[20];

    arg++;

    while ((c = *format++) != 0)
        {
            if (c != '%')
                putchar (c);
            else
                {
                    char *p;

                    c = *format++;
                    switch (c)

```


将名字从 Multiboot standard 变成 multiboot specification
在多重引导头中加入了图形字段
在多种引导信息中加入了 BIOS drive 信息，BIOS 配置表，boot loader 的名字，APM 信息，以及图形信息。
重写了 Textinfo 格式。
使用了更精确的字重写了规范
维护者从 Bryan Ford 以及 Erich Stefan Boleyn 变成了 GNU GRUB

0.6

改变了一些用词
加了头 checksum
将传递给操作系统的机器状态分类

0.5

改变了名字

0.4

做了一些改变，并且加上了规范的 HTML 版本

附录 1 翻译的中英文对照

Boot modules	引导模块
Multiboot	多重引导
Multiboot-complian	兼容多重引导的
Text Segment	正文段
Data Segment	数据段
Stack segment	堆栈段
Multiboot Information struct	多重引导信息结构
Graphic mode	图形模式