第二章

我们采用NASM来作为汇编语言的编译器，GCC作为C语言的编译器，用Makefile作为编译的脚本来实现项目管理，qemu作为我们的虚拟机来运行我们的小程序。各个语言具体语法不介绍了。

首先来说这个系统的进入方式，是通过GRUB用来判断的规范作为我们的指引，这一部分使用汇编来实现，这个程序的名字是任意的，但为了表明它所做的工作，我们将它命名为boot.s。

/boot/boot.s

1. MBOOT\_HEADER\_MAGIC equ 0x1BADB002
2. MBOOT\_PAGE\_ALIGN equ 1 << 0
3. MBOOT\_MEM\_INFO equ 1 << 1
4. MBOOT\_HEADER\_FLAGS equ MBOOT\_PAGE\_ALIGN | MBOOT\_MEM\_INFO
5. MBOOT\_CHECKSUM equ - (MBOOT\_HEADER\_MAGIC + MBOOT\_HEADER\_FLAGS)
6. STACK\_SIZE equ 32768 ; 定义了栈的尺寸大小
8. GLOBAL start ; 内核代码入口
9. GLOBAL glb\_mboot\_ptr ;向外部声明的struct multiboot\*变量
10. EXTERN kern\_entry ; 主函数入口
11. [BITS 32] ; 内核以32位形式编译
12. section .text ; 代码段
13. align 4 ; 4字节对齐
14. Multi\_boot\_header: ; 声明变量，只用Multiboot Header必须的那些部分
15. dd MBOOT\_HEADER\_MAGIC
16. dd MBOOT\_HEADER\_FLAGS
17. dd MBOOT\_CHECKSUM
18. start: ; 汇编的指令开始的地方
19. cli ; 此时还没有设置好保护模式的中断处理，所以必须关闭中断
20. mov esp, stack+STACK\_SIZE ; 设置内核栈地址
21. mov ebp, 0 ; 帧指针修改为 0
22. mov [glb\_mboot\_ptr], ebx ; 将 ebx 中存储的指针存入全局变量
23. call kern\_entry ; 调用内核入口函数，也就是我我们的主函数
24. stop:
25. hlt
26. jmp stop
27. section .bss
29. glb\_mboot\_ptr:
30. resb 4
31. stack:
32. resb STACK\_SIZE

这个部分的代码还是蛮复杂的，里面有些需要说明的地方。

首先，我们的内核是ELF格式的，虽然我们的GRUB并没有做这个要求，但为了方便，所以采用这个格式，也因此对应的有这个格式的一些魔数，规范等。

首先，GRUB有什么要求呢？能够被 GRUB 引导的内核有两个条件：

(1) 需要有一个 Multiboot Header ，这个  Multiboot Header 必须在内核镜像的前 8192 个字节内，并且是首地址是 4 字节对其的。    
(2) 内核的加载地址在 1MB 以上的内存中，这个要求是 GRUB 附加的，并非多重引导规范的规定。

关于Multiboot header，它的分布如下：

|  |  |  |  |
| --- | --- | --- | --- |
| Offset | Type | FieldName | Note |
| 0 | u32 | magic | required |
| 4 | u32 | flags | required |
| 8 | u32 | checksum | required |
| 12 | u32 | header\_addr | if flags[16]isset |
| 16 | u32 | load\_addr | if flags[16]isset |
| 20 | u32 | load\_end\_addr | if flags[16]isset |
| 24 | u32 | bss\_end\_addr | if flags[16]isset |
| 28 | u32 | entry\_addr | if flags[16]isset |
| 32 | u32 | mode\_type | if flags[2]isset |
| 36 | u32 | width | if flags[2]isset |
| 40 | u32 | height | if flags[2]isset |
| 44 | u32 | depth | if flags[2]isset |

其中：

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。自然，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。  
   
    如果设置了 flags 字中的 0 位，所有的引导模块将按页（4KB）边界对齐。有些操作系统能够在启动时将包含引导模块的页直接映射到一个分页的地址空间，因此需要引导模块是页对齐的。  
   
    如果设置了 flags 字中的 1 位，则必须通过 Multiboot 信息结构（参见引导信息格式）的 mem\_\* 域包括可用内存的信息。如果引导程序能够传递内存分布（mmap\_\*域）并且它确实存在，则也包括它。  
   
    如果设置了 flags 字中的 2 位，有关视频模式表（参见引导信息格式）的信息必须对内核有效。   
   
    如果设置了 flags 字中的 16 位，则 Multiboot 头中偏移量 8-24 的域有效，引导程序应该使用它们而不是实际可执行头中的域来计算将 OS 映象载入到那里。如果内核映象为 ELF 格式则不必提供这样的信息，但是如果映象是 a.out 格式或者其他什么格式的话就必须提供这些信息。  
   
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）  
   
header\_addr    
    这里往后的 32 个字节不是必须的，并且对于内核为 ELF 格式时是不需要的，因此就不介绍了。

通过以上内容的介绍，应该能明白之前代码的前5行了。前五行就是定义了这几个规范的格式。

我们分别声明了GRUB的魔数要求，然后又设定了两个数字，分别MBOOT\_PAGE\_ALIGN 和MBOOT\_MEM\_INFO ，这两个数字用来设定FLAG的第0位和第1位，最后我们定义了CHECKSUM。

这里啰嗦一点，文中用到的移位操作，在这种底层设计里很多(我所学尚浅，所以这里行文比较啰嗦)。

其理解为：把某个值向左，或者向右移动指定位数。

举例来说：

1<<0, 就是把1左移0位，实际并无意义，但写在代码里便于理解。

1<<1就是把1左移1位，实际就相当于乘了2，而在FLAG里，由于每一位都有其特定含义，因此这样表示我们就很清楚这个数字是在第二位。

接下来的MBOOT\_PAGE\_ALIGN | MBOOT\_MEM\_INFO就能很清楚明白这是设定了第一位和第二位。

再下一部分7-11行，我们定义了栈的尺寸大小。定义了内核代码入口，multiboot变量，以及主函数的入口kern\_entry。其中**内核代码入口就是我们操作系统开始干活的起点**啦，这部分我们会在我们的链接文件里叙述。而multiboot变量里会存我们系统开始执行的一些重要的信息，需要把这些信息传递出去。

13行[BITS 32]，是说我们的代码要用32位的形式编译。而17行align 4是4字节对齐的含义。至于方括号，以及具体指令的介绍，可以参考NASM的手册。

接下来，19-22行我们声明了我们的一些必须的变量。

再之后，从24行开始，就终于到我们的start了，在上面说过了，start就是我们实际内核的开始。这些工作都是必须的，参考注释简单理解就行。最后我们调用我们30行的主函数。

当我们主函数执行完毕后。回到这里的话，就从32行开始，就hlt停机喽，这里是个无限循环。

而我们的C语言编译的东西，都要在主函数里调用，其它部分基本都是根据规范实现的，而这个主函数就是我们自己的天地啦。

变量段比较简单，在36行到42行，不多说了。

这一部分入口已经定义好了，接下来还需要写一下在这里调用的kern\_entry这个主函数，这里就可以用Ｃ语言实现了。

程序非常简单，起名叫entry.c，如下所示：

/entry.c

1. #include "types.h"
2. int kern\_entry()
3. {
4. uint8\_t \*input = (uint8\_t \*)0xB8000;
5. uint8\_t color = (0 << 4) | (15 & 0x0F);
6. \*input++ = 'H'; \*input++ = color;
7. \*input++ = 'e'; \*input++ = color;
8. \*input++ = 'l'; \*input++ = color;
9. \*input++ = 'l'; \*input++ = color;
10. \*input++ = 'o'; \*input++ = color;
11. \*input++ = ','; \*input++ = color;
12. \*input++ = ' '; \*input++ = color;
13. \*input++ = 'K'; \*input++ = color;
14. \*input++ = 'e'; \*input++ = color;
15. \*input++ = 'r'; \*input++ = color;
16. \*input++ = 'n'; \*input++ = color;
17. \*input++ = 'e'; \*input++ = color;
18. \*input++ = 'l'; \*input++ = color;
19. \*input++ = '!'; \*input++ = color;
20. return 0;
21. }

这一部分和我们写过的最简单的Hello world程序区别不大，唯一的是我们的入口主函数变成了kern\_entry()，而为了让我们的程序显示一点内容，我们让他输出一个hello, kernel! 但这个怪异的写法在这里就不深究了，后面讲到打印输出的时候就会明白了。

其中颜色可以自己设置哦，uint8\_t color = (0 << 4) | (15 & 0x0F) 这句话中，前面左移4位的0表示黑底，后面和0F(1111)取交的是15，代表了白字，尝试改变这两个数字从0到15，看看有多少种颜色~。

而这里第1行中的库，是我们为了统一，自己重新定义的格式，代码如下：

/include/types.h

1. #ifndef INCLUDE\_TYPES\_H\_
2. #define INCLUDE\_TYPES\_H\_
3. #ifndef NULL
4. #define NULL 0
5. #endif
6. #ifndef TRUE
7. #define TRUE 1
8. #define FALSE 0
9. #endif
10. typedef unsigned int uint32\_t;
11. typedef int int32\_t;
12. typedef unsigned short uint16\_t;
13. typedef short int16\_t;
14. typedef unsigned char uint8\_t;
15. typedef char int8\_t;
16. #endif

这样重新定义的好处大大的，如果一旦我们的需要在其它平台使用，只需要在这里对我们的定义的类型重新设置就可以了~极大增加了可移植性。而开始的#ifndef这些，也是防止类型冲突所设定的部分，在后面我们写的库文件里都会用到。

好了。终于，我们有了第一部分的代码了。应该编译运行了，我们用Makefile来实现。

Makefile是linux下自动编译的一个脚本，根据规则写好编译的方式，在shell里直接输入Makefile就能对我们的文件批量进行编译。

我们的Makefile脚本如下：

/Makefile

1. C\_SOURCES = $(shell find . -name "\*.c")
2. C\_OBJECTS = $(patsubst %.c, %.o, $(C\_SOURCES))
3. S\_SOURCES = $(shell find . -name "\*.s")
4. S\_OBJECTS = $(patsubst %.s, %.o, $(S\_SOURCES))
5. CC = gcc
6. LD = ld
7. ASM = nasm
8. C\_FLAGS = -c -Wall -m32 -ggdb -gstabs+ -nostdinc -fno-builtin -fno-stack-protector -I include
9. LD\_FLAGS = -T scripts/kernel.ld -m elf\_i386 -nostdlib
10. ASM\_FLAGS = -f elf -g -F stabs
11. all: $(S\_OBJECTS) $(C\_OBJECTS) link update\_image
12. # The automatic variable `$<' is just the first prerequisite
13. .c.o:
14. @echo 编译代码文件 $< ...
15. $(CC) $(C\_FLAGS) $< -o $@
16. .s.o:
17. @echo 编译汇编文件 $< ...
18. $(ASM) $(ASM\_FLAGS) $<
19. link:
20. @echo 链接内核文件...
21. $(LD) $(LD\_FLAGS) $(S\_OBJECTS) $(C\_OBJECTS) -o jf\_kernel
22. .PHONY:clean
23. clean:
24. $(RM) $(S\_OBJECTS) $(C\_OBJECTS) jf\_kernel
25. .PHONY:update\_image
26. update\_image:
27. sudo mount floppy.img /mnt/kernel
28. sudo cp jf\_kernel /mnt/kernel/jf\_kernel
29. sleep 1
30. sudo umount /mnt/kernel
31. .PHONY:mount\_image
32. mount\_image:
33. sudo mount floppy.img /mnt/kernel
34. .PHONY:umount\_image
35. umount\_image:
36. sudo umount /mnt/kernel
37. .PHONY:qemu
38. qemu:
39. qemu -fda floppy.img -boot a
40. .PHONY:debug
41. debug:
42. qemu -S -s -fda floppy.img -boot a &
43. sleep 1
44. cgdb -x scripts/gdbinit

我给自己的内核起名加jf\_kernel，如果需要修改，需要把Makefile和floppy.img里面GRUB的配置文件修改。

最后，还有我们的链接文件，起名叫kernel.ld，代码如下：

/scripts/kernel.ld

1. ENTRY(start)
2. SECTIONS
3. {
4. . = 0x100000;
5. .text :
6. {
7. \*(.text)
8. . = ALIGN(4096);
9. }
10. .data :
11. {
12. \*(.data)
13. \*(.rodata)
14. . = ALIGN(4096);
15. }
16. .bss :
17. {
18. \*(.bss)
19. . = ALIGN(4096);
20. }
21. .stab :
22. {
23. \*(.stab)
24. . = ALIGN(4096);
25. }
26. .stabstr :
27. {
28. \*(.stabstr)
29. . = ALIGN(4096);
30. }
32. /DISCARD/ : { \*(.comment) \*(.eh\_frame) }
33. }

此时我们的文件目录结构如下：

jf@ubuntu:~/OS\_jf$ tree

.

├── boot

│   └── boot.s

├── entry.c

├── floppy.img

├── include

│   └── types.h

├── Makefile

└── scripts

└── kernel.ld

3 directories, 6 files

然后我们执行 make和make qemu就可以看到我们的hello kernel！啦！

jf@ubuntu:~/OS\_jf$ make

编译汇编文件 boot/boot.s ...

nasm -f elf -g -F stabs boot/boot.s

编译代码文件 entry.c ...

gcc -c -Wall -m32 -ggdb -gstabs+ -nostdinc -fno-builtin -fno-stack-protector -I include entry.c -o entry.o

链接内核文件...

ld -T scripts/kernel.ld -m elf\_i386 -nostdlib ./boot/boot.o ./entry.o -o jf\_kernel

sudo mount floppy.img /mnt/kernel

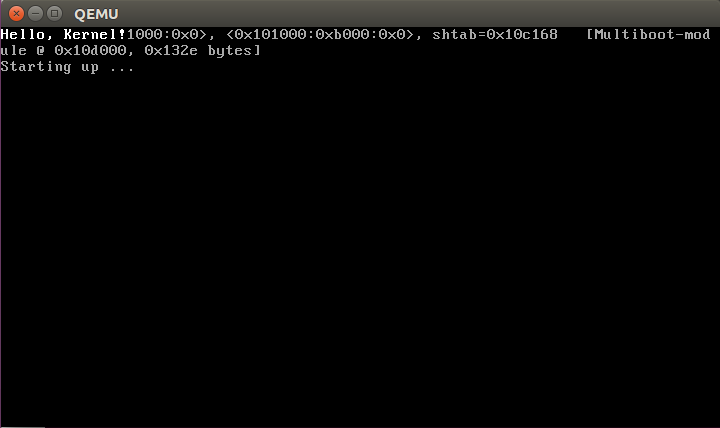
sudo cp jf\_kernel /mnt/kernel/jf\_kernel

sleep 1

sudo umount /mnt/kernel

jf@ubuntu:~/OS\_jf$ make qemu

结果如下：



在grub载入内核后，它会设置一些状态。有这些：

1. CS 指向基地址为 0x00000000，限长为4G – 1的代码段描述符。
2. DS，SS，ES，FS 和 GS 指向基地址为0x00000000，限长为4G–1的数据段描述  
   符。
3. A20 地址线已经打开。
4. 页机制被禁止。
5. 中断被禁止。
6. EAX = 0x2BADB002
7. 系统信息和启动信息块的线性地址保存在 EBX中(相当于一个指针)

这部分内容后续会继续用到。