第五章

X86处理器有多种模式，这些内容可以在计算机组成原理这类书中学到，不赘述了。而我们此前所为，都是在实模式下进行的。

实模式有诸多弊病，在实模式下，用户的权限太高了，我们可以在任何地方写入我们的数据，很可能一不小心就把我们的操作系统，软件端口等内容覆盖，造成难以估计的后果。同时，在实模式下，我们只能使用16位的寄存器，也就是说，我们能使用的地址空间仅仅有64KB大小。也没有内存保护，没有多任务，运行的内容也没有分级，一个简单的cli或者hlt指令就可能让整个系统崩溃。

在早期，为了让计算机能够使用更大的内存，引入了分段的概念，按照最大64Kb把内存分段。并为了在后期让80286与8086/8088在实模式下行为一致，又搞出了个A20 gate，可谓是煞费苦心。(在第二章最后提到过这个A20地址线)

后面到了386时代，开始有了保护模式，这个时候对内存的管理是按照**分页(paging)**来管理的。但为了与前面的内容兼容，此时的分段还是必须的。一直到了2003年开始的x86-64，这一堪称爷爷辈的分段概念才逐渐开始被淘汰。

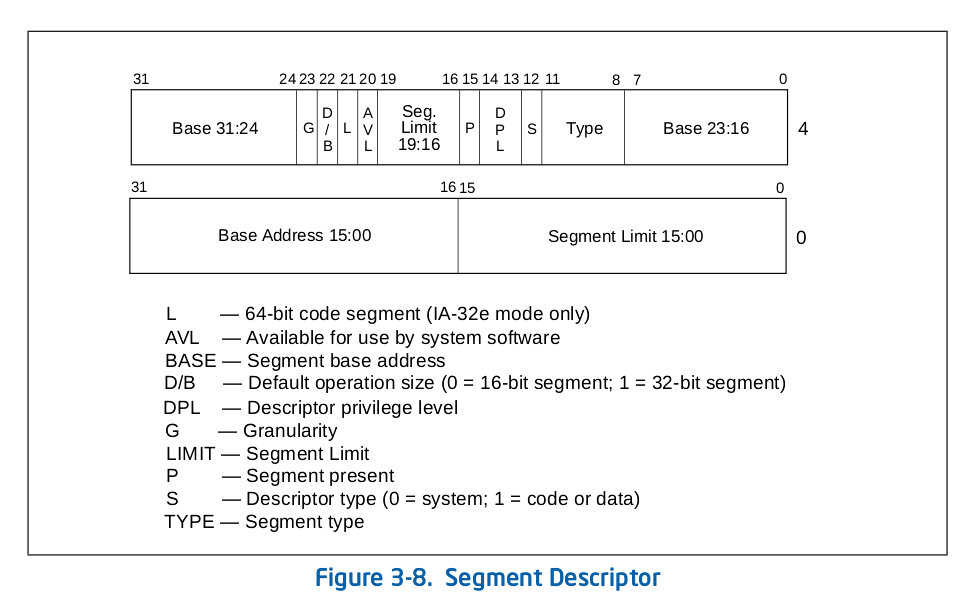
那为什么要分段呢，其实很简单。因为早期的寄存器和ALU只有16位，而地址线有20位。为了遍历更大的地址，把内存分段，用基址:变址(实模式是段:偏移)来遍历更大的地址。

这部分无须深究，简单理解就好，明白在32位保护模式下分段是如何实现的。以及它带来的好处和用处就行。

由于32位的地址线已经足够遍历4G的内存空间了。所以分段实在是有些鸡肋，这时候一个聪明的方法就是干脆把内存直接分成一个段，也就是所谓的**平坦模式(flat mode)**。

在32位下，我们用**8个字节大小**的内容来描述一段内存，并且把这个描述的小东东称之为**段描述符(Segment Descriptor)**，而描述符的集合，就是我们的**描述符表(descriptor table)**，这个集合表有三类，包括**Global Descriptor Table (GDT)**, **Local Descriptor Table (LDT)**, 和 **Interrupt Descriptor Table (IDT)**; 对于它们的操作，都是在**ring 0**权限下进行的。并且CPU中有特定的寄存器，来存储它们的基址，分别是**GDTR, LDTR**和**IDTR** 。

我们先来看看GDT，也就是全局描述符表。



上面是Intel的官方文档的图，我这里详细解释如下：

**Bits 56-63:** Bits 24-32 of the base address

**Bit 55:** Granularity

**0:** None

**1:** Limit gets multiplied by 4K

**Bit 54:** Segment type

**0:** 16 bit

**1:** 32 bit

**Bit 53:** 64-bit code

**Bits 52:** Reserved for OS use

**Bits 48-51:** Bits 16-19 of the segment limit

**Bit 47** Segment is in memory (Used with Virtual Memory)

**Bits 45-46:** Descriptor Privilege Level

**0:** (Ring 0) Highest

**3:** (Ring 3) Lowest

**Bit 44:** Descriptor Bit

**0:** System Descriptor

**1:** Code or Data Descriptor

**Bits 41-43:** Descriptor Type

**Bit 43:** Executable segment

**0:** Data Segment

**1:** Code Segment

**Bit 42:** Expansion direction (Data segments), conforming (Code Segments)

**Bit 41:** Readable and Writable

**0:** Read only (Data Segments); Execute only (Code Segments)

**1:** Read and write (Data Segments); Read and Execute (Code Segments)

**Bit 40:** Access bit (Used with Virtual Memory)

**Bits 16-39:** Bits 0-23 of the Base Address

**Bits 0-15:** Bits 0-15 of the Segment Limit

我们前面说到，管理这个表的方式是通过一个特定的寄存器**GDTR**来实现的。这个寄存器全称叫**全局描述符表寄存器**。共**48位**，0-15位表示GDT的大小，而16-47位是GDT的首地址。(所以既然大小是16位，那么2的16次方就是65536字节，除以每一个描述符的8字节，那么最多能创建8192个描述符).

看一下这一部分的代码吧：

/include/gdt.h

1. #ifndef INCLUDE\_GDT\_H\_
2. #define INCLUDE\_GDT\_H\_
3. #include "types.h"
4. // 全局描述符类型
5. typedef
6. struct g\_descriptor {
7. uint16\_t limit\_low; // 段界限 0~15
8. uint16\_t base\_low; // 段基地址 16~31
9. uint8\_t base\_middle; // 段基地址 32~39
10. uint8\_t access; // 段存在位、描述符特权级、描述符类型、描述符子类别
11. uint8\_t granularity; // 其他标志、段界限 19～16
12. uint8\_t base\_high; // 段基地址 56~63
13. } \_\_attribute\_\_((packed)) g\_descriptor;
14. // GDTR
15. typedef
16. struct gdtr{
17. uint16\_t limit; // 全局描述符表限长
18. uint32\_t base; // 全局描述符表 32位 基地址
19. } \_\_attribute\_\_((packed)) gdtr;
20. // 初始化全局描述符表
21. void init\_gdt();
22. // GDT 加载到 GDTR 的函数[汇编实现]
23. extern void gdt\_flush(uint32\_t);
24. #endif // INCLUDE\_GDT\_H\_

我们分别定义了**全局描述符(global descriptor)**和**gdtr**的结构，并声明了两个函数，一个用来初始化gdt，一个用来加载gdtr。

那么我们要如何初始化呢？按照之前说的，根据这个全局描述符，把4G内存映射到上来，然后用gdtr记录我们的gdt，加载就行了。这样的话我们的GDT相当于只有一个descriptor。一般来讲，是至少需要设置3个descriptor，第一个是NULL descr，第二个是code descriptor，第三个是data descriptor。

我们这里设置五个，增加一个用户的code descriptor和用户的data descriptor。

/gdt/gdt.c

1. #include "print.h"
2. #include "gdt.h"
3. // 全局描述符表个数
4. #define GDT\_LENGTH 5
5. // 全局描述符表定义
6. g\_descriptor gdt[GDT\_LENGTH];
7. // GDTR
8. gdtr gdt\_ptr;
9. // 全局描述符表构造函数，根据下标构造
10. static void gdt\_set\_gate(int32\_t num, uint32\_t base, uint32\_t limit, uint8\_t access, uint8\_t gran);
11. // 声明内核栈地址
12. extern uint32\_t stack;
13. // 初始化全局描述符表
14. void init\_gdt()
15. {
16. // 全局描述符表界限 e.g. 从 0 开始，所以总长要 - 1
17. gdt\_ptr.limit = sizeof(g\_descriptor) \* GDT\_LENGTH - 1;
18. gdt\_ptr.base = (uint32\_t)&gdt;
19. // 采用 Intel 平坦模型
20. gdt\_set\_gate(0, 0, 0, 0, 0); // 按照 Intel 文档要求，第一个描述符必须全 0
21. gdt\_set\_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // 指令段
22. gdt\_set\_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // 数据段
23. gdt\_set\_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // 用户模式代码段
24. gdt\_set\_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // 用户模式数据段
25. // 加载全局描述符表地址到 GPTR 寄存器
26. gdt\_flush((uint32\_t)&gdt\_ptr);
27. }
28. static void gdt\_set\_gate(int32\_t num, uint32\_t base, uint32\_t limit, uint8\_t access, uint8\_t gran)
29. {
30. gdt[num].base\_low = (base & 0xFFFF);
31. gdt[num].base\_middle = (base >> 16) & 0xFF;
32. gdt[num].base\_high = (base >> 24) & 0xFF;
33. gdt[num].limit\_low = (limit & 0xFFFF);
34. gdt[num].granularity = (limit >> 16) & 0x0F;
35. gdt[num].granularity |= gran & 0xF0;
36. gdt[num].access = access;
37. }

这部分代码也不复杂，主要在于具体的设置gdt的数值。结合intel文档，我来简单说说：

首先我们使用了平坦模式，所以基址都是0，limit都是0xFFFFFFFF。

先看access的值，对应40~47位

0x9A：1001 1010

0x92：1001 0010

0xFA：1111 1010

0xF2：1111 0010

回头看看我们前面的说明，

40位表示是否可以用虚拟内存，这里都是0。

41位表示可读还是可写，我们这里都是有完全权限。

42位表示扩展方向，这里都是一样的。

43位表示表示是否可以执行，代码段是1，数据段是0.

44位表示descriptor的种类，这里都是1，表示为代码或者数据段，而不是系统段。

45位~46位是说我们的级别。这里前两个是系统的代码段和数据段，级别为ring 0，所以00。

后两个是用户段，级别ring 3，所以为11。

47位是在使用虚拟内存时是否在内存存储我们的段，这里都是存储。

再来看Granularity，对应48~55位。分别由limit和gran设置。

48~51位，由 (limit >> 16) & 0x0F设置了，表示段界限的16-19位。设置为了1111，

52~55位，gran & 0xF0。gran的值为0xCF：1100 1111。这里用了高4位，也就是1100

52位，保留，这里为0

53位，64位会用到，这里是0

54位，表示段的类型，都是32位的，所以是1.

55位，表示粒度，这里都为1，表示4K对齐

看到这里，这部分代码算是解释清楚了。

把gdt和gdtr都写好了。就差往寄存器里面装了。这部分我们直接用汇编实现：

/gdt/gdt\_s.s

1. [GLOBAL gdt\_flush]
2. gdt\_flush:
3. mov eax, [esp+4] ; 参数存入 eax 寄存器
4. lgdt [eax] ; 加载到 GDTR [修改原先GRUB设置]
5. mov ax, 0x10 ; 加载我们的数据段描述符
6. mov ds, ax ; 更新所有可以更新的段寄存器
7. mov es, ax
8. mov fs, ax
9. mov gs, ax
10. mov ss, ax
11. jmp 0x08:.flush ; 远跳转，0x08是我们的代码段描述符
12. ; 远跳目的是清空流水线并串行化处理器
13. .flush:
14. ret

还记得上一章说到的，函数如何传递参数的吗。这里调用函数，就会把我们gdt\_flush((uint32\_t)&gdt\_ptr)里面的参数压栈，所以[esp+4] 就是我们gdt\_ptr的东西了。

然后我们使用lgdt这个汇编指令来加载我们的GDT，注意这个指令是ring 0的指令。

后面的内容我们更新了寄存器。在实模式中，我们的寻址是**Segment:Offset**，而在保护模式中，是采用了 **Descriptor:Address**的寻址模式，所以我们要更新我们的寄存器才行。mov ax, 0x10是说加载我们的数据段描述符，因为在段描述符里，基址的位置就是16位的偏移（你不会不知道0x10是16吧2333）。因为在执行lgdt之后，处理器处理的地址都会按照gdt的基址开始。

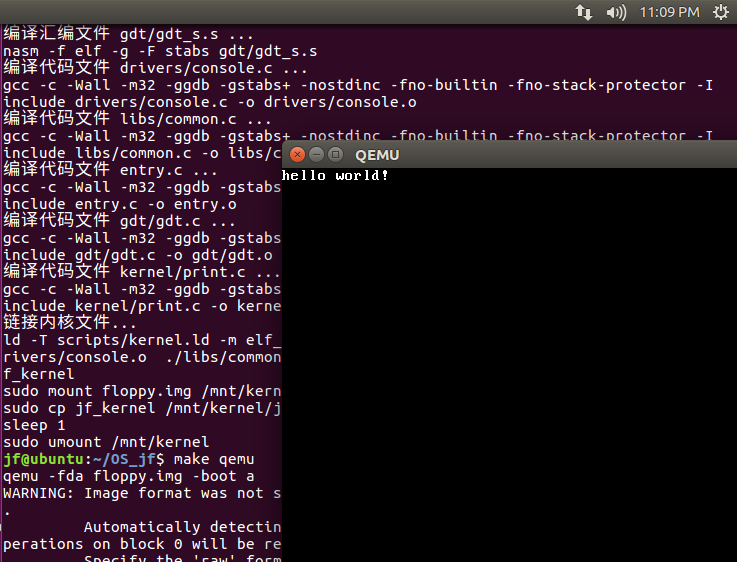
后面的jmp 0x08:.flush可能会让你诧异，因为它就是下一行代码，但我们还是用了jmp来跳一下，这是为何？当然有深意了，第一，Intel不允许直接修改段寄存器cs的值，我们只好这样通过这种方式更新cs段寄存器；第二，x86以后CPU所增加的指令流水线和高速缓存可能会在新的全局描述符表加载之后依旧保持之前的缓存，那么修改GDTR之后最安全的做法就是立即清空流水线和更新高速缓存。而0x08正是我们的descriptor大小。

最后修改下我们的entry

/entry.c

1. #include"console.h"
2. #include"print.h"
3. #include“gdt.h”
4. int kern\_entry()
5. {
6. init\_gdt();
7. console\_clear();
8. printk("hello world!\n");
9. return 0;
10. }

执行结果如下：



如果和之前的结果一样，那就说明没有问题。因为lgdt指令是ring 0的，而我们的gdt一旦设置有误，那么就会造成triple fault。没有发生就说明一些正常。

此时的目录：

jf@ubuntu:~/OS\_jf$ tree

.

├── boot

│   └── boot.s

├── drivers

│   └── console.c

├── entry.c

├── floppy.img

├── gdt

│   ├── gdt.c

│   └── gdt\_s.s

├── include

│   ├── common.h

│   ├── console.h

│   ├── gdt.h

│   ├── print.h

│   ├── string.h

│   ├── types.h

│   └── vargs.h

├── kernel

│   └── print.c

├── libs

│   ├── common.c

│   └── string.c

├── Makefile

└── scripts

├── gdbinit

└── kernel.ld

7 directories, 19 files

<http://www.brokenthorn.com/Resources/OSDev8.html>

<https://en.wikipedia.org/wiki/X86_memory_segmentation>

http://www.mouseos.com/arch/mechanism.html