第六章

在这一章，我们来实现中断(interrupt)。

所谓中断，是一个外部的异步信号，它提供了一种中断当前任务来执行其他任务的一种方式。此外，它还有帮助解决问题的功能，当处理器发现执行的代码有错误的时候，中断会给处理器一种修复或者处理问题的手段。比如我们用一个数去除0，就会引发一个中断。

中断分为硬件中断和软件中断，由中断的产生者来区分。

硬件中断(也叫中断请求，interrupt request，IRQ)是由芯片外部产生的，通过向CPU的INTR管脚或者其它等效的管脚发送信号，来获得相应的处理。可以分为两种：

中断请求线/基于引脚的中断请求(*IRQ Lines, or Pin-based IRQs*)：这些中断是芯片上静态排列的。它们统一连到中断请求控制器(IRQ controller)上，中断请求控制器把它们序列化，依次发送到CPU上。最典型的控制器就是Intel 的8259可编程中断控制器了(PIC)，最早的IBM PC用的就是这个，一个8259 PIC可以提供8个IRQ。现代计算机已经把这个芯片集成到了南桥中了(并且有了APIC, Advanced Programmable Interrupt Controller来代替PIC)。

基于信息的中断：这个是一种特殊的消息传递的方法，不需要额外占用线路，通过向特定的地址发送关于中断设备，中断以及向量信息(会有一个中断向量表，Interrupt Vector Table，IVT的东西，后面介绍)。设备会分配一个地址，这个位置可以通过固件或者内核软件使用。就可以通过使用特定的设备总线的协议产生设备的中断，一个典型的例子就是PCI总线。

软件中断：这个是在运行的软件向CPU发出的一个信号，这种中断一般用作系统调用，在x86上，启动软件中断的指令是INT，相信用过汇编的对这个一定很熟悉了。在x86上，中断向量一般有256个，除了一些预留的以外，还有很多可以选择的，但大部分的情况都是用0x80这个向量作为系统调用的中断。很熟悉，是吧。

下面我们来实现中断。

对于中断的定义，和上一章的分段一样，也有一个中断表来维护。

# 实模式

在实模式下，我们的中断表叫做IVT，Interrupt Vector Table。这个表是一系列的中断向量，总共256个。这个向量表放置在物理内存的前1024字节处，从地址0x0直到0x3FF，每个向量大小都是4字节，包括以下内容：

字节0：偏移中断程序的低地址(IR)

字节1：IR的高地址

字节2：IR所在段的低位地址

字节3：IR所在段的的高地址

中断处理程序(Interrupt Routines，IRs)，是一个特殊的函数，用来处理中断请求(IRQ)，当处理器执行中断时，比如INT，就会按照中断表来找特定的中断处理函数(IR)。

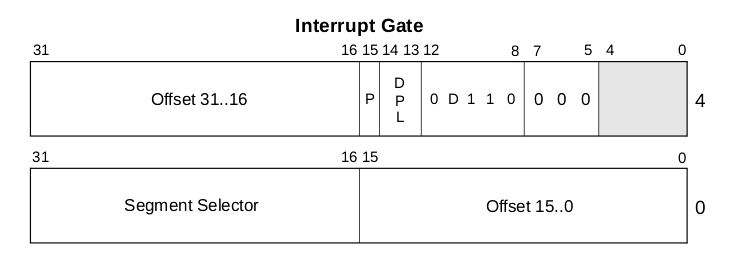
中断服务程序(Interrupt Service Routines，ISRs)，是一个中断处理程序，是硬件响应中断而调用的软件程序。 ISR检查中断并确定如何处理它。ISR其实就是IR，是我们为处理硬件和软件中断定义的调用方法。有三种类型，一级中断处理程序(First Level Interrupt Handler，FLIH)，二级中断处理程序(Second Level Interrupt Handler，SLIH)和嵌套中断处理程序(Nested Interrupt Handlers)

对于IVT来讲，只需要存放好IR的地址就行，我们可以把IR放在地址的任意位置，把IVT设置好这个地址就行。下面给出IVT的表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x86 Interrupt Vector Table (IVT)** | | | | |
| Base Address | Interrupt Number | Description |  |  |
| 0x000 | 0 | Divide by 0 | | | |
| 0x004 | 1 | Single step (Debugger) | | | |
| 0x008 | 2 | Non Maskable Interrupt (NMI) Pin | | | |
| 0x00C | 3 | Breakpoint (Debugger) | | | |
| 0x010 | 4 | Overflow | | | |
| 0x014 | 5 | Bounds check | | | |
| 0x018 | 6 | Undefined Operation Code (OPCode) instruction | | | |
| 0x01C | 7 | No coprocessor | | | |
| 0x020 | 8 | Double Fault | | | |
| 0x024 | 9 | Coprocessor Segment Overrun | | | |
| 0x028 | 10 | Invalid Task State Segment (TSS) | | | |
| 0x02C | 11 | Segment Not Present | | | |
| 0x030 | 12 | Stack Segment Overrun | | | |
| 0x034 | 13 | General Protection Fault (GPF) | | | |
| 0x038 | 14 | Page Fault | | | |
| 0x03C | 15 | Unassigned | | | |
| 0x040 | 16 | Coprocessor error | | | |
| 0x044 | 17 | Alignment Check (486+ Only) | | | |
| 0x048 | 18 | Machine Check (Pentium/586+ Only) | | | |
| 0x05C | 19-31 | Reserved exceptions | | | |
| 0x068 - 0x3FF | 32-255 | Interrupts free for software use | | | |

# 保护模式

在保护模式下，我们的中断表叫做中断描述符表(The Interrupt Descriptor Table，IDT)和之前说的GDT颇为类似。中断描述符是8字节大小，图示如下：



前面我们有简单说到中断的用处，在这里我们的Interrupt Descrioptor按照功能可以分为三类：

Interrupt Gate，中断门。用于指定中断服务程序(interrupt service routine)。

Trap Gate，陷阱门。由中断或者异常激发，和中断门几乎一样，除了type位的区别。

Task Gate任务门。用来产生任务的切换，IDT中的任务门引用GDT中的TSS描述符，以与普通任务切换相同的方式来处理对处理程序任务的切换。

详细说一下每位的含义：

Bits 0...15:

**Interrupt / Trap Gate:** Offset address Bits 0-15 of IR

**Task Gate:** Not used.

Bits 16...31:

**Interrupt / Trap Gate:** Segment Selector (Useually 0x10)

**Task Gate:** TSS Selector

Bits 31...35: Not used

Bits 36...38:

**Interrupt / Trap Gate:** Reserved. Must be 0.

**Task Gate:** Not used.

Bits 39...43:

**Interrupt Gate:** Of the format 0D110, where D determins size

**01110** - 32 bit descriptor

**00110** - 16 bit descriptor

**Task Gate:** Must be 00101

**Trap Gate:** Of the format 0D111, where D determins size

**01111** - 32 bit descriptor

**00111** - 16 bit descriptor

Bits 44...45: Descriptor Privedlge Level (DPL)

**00:** Ring 0

**01:** Ring 1

**10:** Ring 2

**11:** Ring 3

Bit 46: Segment is present (1: Present, 0:Not present)

Bits 47...62:

**Interrupt / Trap Gate:** Bits 16...31 of IR address

**Task Gate:** Not used

讲了这么多了，先给出我们的idt的结构吧：

/include/idt.h

1. #ifndef INCLUDE\_IDT\_H\_
2. #define INCLUDE\_IDT\_H\_
3. #include "types.h"
4. // 初始化中断描述符表
5. void init\_idt();
6. // 中断描述符
7. typedef
8. struct i\_descriptor {
9. uint16\_t base\_lo; // 中断处理函数地址 15～0 位
10. uint16\_t sel; // 目标代码段描述符选择子
11. uint8\_t always0; // 置 0 段
12. uint8\_t flags; // 一些标志，文档有解释
13. uint16\_t base\_hi; // 中断处理函数地址 31～16 位
14. }\_\_attribute\_\_((packed)) i\_descriptor;
15. // IDTR
16. typedef
17. struct idtr{
18. uint16\_t limit; // 限长
19. uint32\_t base; // 基址
20. } \_\_attribute\_\_((packed)) idtr;

*(代码后续还有)*

这部分内容可不少，部分内容参考上一章的GDT应该可以明白。

10行开始就是我们的中断描述符的结构体了，20行开始是我们的IDTR的结构，和GDT那里的GDTR一样，不多说了。

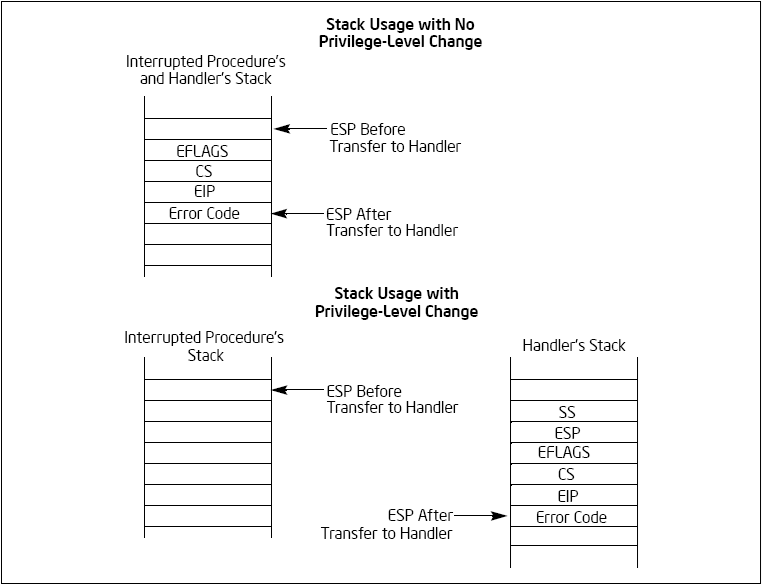
前面对中断有了很详细的介绍了。具体中断的执行，是这样的：

CPU在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中是否发送中断请求过来，当触发了中断或者异常时，处理器使用异常或中断号作为IDT的索引。处理器执行计算IDTR.baseAddress + index \* 8(8是8字节大小)，index是中断号，处理器根据这个检索中断处理程序的描述符索引的基址。

分两种情况：

1. 如果处理程序将在较低特权级别(描述符的第42-45位)执行，则会发生堆栈切换，执行以下内容：
   * 对于当前要执行的任务，处理程序从TSS获得当前栈要用到的段选择子和栈指针的信息。
   * 处理器把当前的状态，包括EFLAGS，CS和EIP存到新的栈上
   * 如果异常产生了错误代码，那么把该错误代码push到新的栈上
2. 如果处理程序将以相同的权限级别执行(当前特权级别(current privilege level，CPL)
   * 处理器把当前的状态，包括EFLAGS，CS和EIP存到当前栈上
   * 如果异常产生了错误代码，那么把该错误代码push到当前栈EIP的后面

图示为这两种情况下栈的样子:



错误代码的每一位都有自己的含义，格式如下：

Bit 0: External event

0: Internal or software event triggered the error.

1: External or hardware event triggered the error.

Bit 1: Description location

0: Index portion of error code refers to descriptor in GDT or current LDT.

1: Index portion of error code refers to gate descriptor in IDT.

Bit 2: GDT/LDT. Only use if the descriptor location is 0.

0: This indicates the index portion of the error code refers to a descriptor in the current GDT.

1: This indicates the index portion of the error code refers to a segment or gate descriptor in the LDT.

Bits 3-15: Segment selector index. This is an index into the IDT, GDT, or current LDT to the segment or gate selector bring refrenced by the error code.

Bits 16-31: Reserved

但注意在外部产生的异常(通过INTR，LINT0，LINT1引脚)或INT n指令产生异常时，错误代码不会被压入堆栈。同时，页面错误异常错误的错误代码格式与这个不同。

当中断执行结束后，中断处理程序必须使用IRET或者IRETD来返回。

所以，我们所要做的工作，不止是搞一个中断描述符表，还需要自己来写栈切换，寄存器保护和中断处理函数等内容。按照Intel的规定，0～19号中断属于CPU所有，而且第20-31号中断也被Intel保留，所以从32～255号才属于用户自定义中断。虽说是"用户自定义"，其实在x86上有些中断按照习惯还是给予了固定的设备。比如32号是timer中断，33号是键盘中断等等。

/include/idt.h

1. // 寄存器类型
2. typedef
3. struct pt\_regs\_t {
4. uint32\_t ds; // 用于保存用户的数据段描述符
5. uint32\_t edi; // 从 edi 到 eax 由 pusha 指令压入
6. uint32\_t esi;
7. uint32\_t ebp;
8. uint32\_t esp;
9. uint32\_t ebx;
10. uint32\_t edx;
11. uint32\_t ecx;
12. uint32\_t eax;
13. uint32\_t int\_no; // 中断号
14. uint32\_t err\_code; // 错误代码(有中断错误代码的中断会由CPU压入)
15. uint32\_t eip; // 以下由处理器自动压入
16. uint32\_t cs;
17. uint32\_t eflags;
18. uint32\_t useresp;
19. uint32\_t ss;
20. } pt\_regs;
21. // 定义中断处理函数指针
22. typedef void (\*interrupt\_handler\_t)(pt\_regs \*);
23. // 注册一个中断处理函数
24. void register\_interrupt\_handler(uint8\_t n, interrupt\_handler\_t h);
25. // 调用中断处理函数
26. void isr\_handler(pt\_regs \*regs);
27. // 声明中断处理函数 0-19 属于 CPU 的异常中断
28. // ISR:中断服务程序(interrupt service routine)
29. void isr0(); // 0 #DE 除 0 异常
30. void isr1(); // 1 #DB 调试异常
31. void isr2(); // 2 NMI
32. void isr3(); // 3 BP 断点异常
33. void isr4(); // 4 #OF 溢出
34. void isr5(); // 5 #BR 对数组的引用超出边界
35. void isr6(); // 6 #UD 无效或未定义的操作码
36. void isr7(); // 7 #NM 设备不可用(无数学协处理器)
37. void isr8(); // 8 #DF 双重故障(有错误代码)
38. void isr9(); // 9 协处理器跨段操作
39. void isr10(); // 10 #TS 无效TSS(有错误代码)
40. void isr11(); // 11 #NP 段不存在(有错误代码)
41. void isr12(); // 12 #SS 栈错误(有错误代码)
42. void isr13(); // 13 #GP 常规保护(有错误代码)
43. void isr14(); // 14 #PF 页故障(有错误代码)
44. void isr15(); // 15 CPU 保留
45. void isr16(); // 16 #MF 浮点处理单元错误
46. void isr17(); // 17 #AC 对齐检查
47. void isr18(); // 18 #MC 机器检查
48. void isr19(); // 19 #XM SIMD(单指令多数据)浮点异常
49. // 20-31 Intel 保留
50. void isr20();
51. void isr21();
52. void isr22();
53. void isr23();
54. void isr24();
55. void isr25();
56. void isr26();
57. void isr27();
58. void isr28();
59. void isr29();
60. void isr30();
61. void isr31();
62. // 32～255 用户自定义异常
63. void isr255();
64. #endif // INCLUDE\_IDT\_H\_

由于中断处理函数中，除了除了CPU本身保护的现场外，其它寄存器的保存和恢复过程都是一样的。所以，如果在每个中断处理函数中都实现一次显然冗余而且易错。所以我们很自然把原本的中断处理函数逻辑上拆解为三部分，第一部分是一致的现场保护操作；第二部分是每个中断特有的处理逻辑；第三部分又是一致的现场恢复。实际上我们把每个中断处理函数拆解为四段，在四个函数里实现。具体的实现如下：

/idt/idt\_s.s

1. [GLOBAL idt\_flush]
2. idt\_flush:
3. mov eax, [esp+4] ; 参数存入 eax 寄存器
4. lidt [eax] ; 加载到 IDTR
5. ret
6. .end:
7. ; 定义两个构造中断处理函数的宏(有的中断有错误代码，有的没有)
8. ; 用于没有错误代码的中断
9. %macro ISR\_NOERRCODE 1
10. [GLOBAL isr%1]
11. isr%1:
12. cli ; 首先关闭中断
13. push 0 ; push 无效的中断错误代码(起到占位作用，便于所有isr函数统一清栈)
14. push %1 ; push 中断号
15. jmp isr\_common\_stub
16. %endmacro
17. ; 用于有错误代码的中断
18. %macro ISR\_ERRCODE 1
19. [GLOBAL isr%1]
20. isr%1:
21. cli ; 关闭中断
22. push %1 ; push 中断号
23. jmp isr\_common\_stub
24. %endmacro
25. ; 定义中断处理函数
26. ISR\_NOERRCODE 0 ; 0 #DE 除 0 异常
27. ISR\_NOERRCODE 1 ; 1 #DB 调试异常
28. ISR\_NOERRCODE 2 ; 2 NMI
29. ISR\_NOERRCODE 3 ; 3 BP 断点异常
30. ISR\_NOERRCODE 4 ; 4 #OF 溢出
31. ISR\_NOERRCODE 5 ; 5 #BR 对数组的引用超出边界
32. ISR\_NOERRCODE 6 ; 6 #UD 无效或未定义的操作码
33. ISR\_NOERRCODE 7 ; 7 #NM 设备不可用(无数学协处理器)
34. ISR\_ERRCODE 8 ; 8 #DF 双重故障(有错误代码)
35. ISR\_NOERRCODE 9 ; 9 协处理器跨段操作
36. ISR\_ERRCODE 10 ; 10 #TS 无效TSS(有错误代码)
37. ISR\_ERRCODE 11 ; 11 #NP 段不存在(有错误代码)
38. ISR\_ERRCODE 12 ; 12 #SS 栈错误(有错误代码)
39. ISR\_ERRCODE 13 ; 13 #GP 常规保护(有错误代码)
40. ISR\_ERRCODE 14 ; 14 #PF 页故障(有错误代码)
41. ISR\_NOERRCODE 15 ; 15 CPU 保留
42. ISR\_NOERRCODE 16 ; 16 #MF 浮点处理单元错误
43. ISR\_ERRCODE 17 ; 17 #AC 对齐检查
44. ISR\_NOERRCODE 18 ; 18 #MC 机器检查
45. ISR\_NOERRCODE 19 ; 19 #XM SIMD(单指令多数据)浮点异常
46. ; 20~31 Intel 保留
47. ISR\_NOERRCODE 20
48. ISR\_NOERRCODE 21
49. ISR\_NOERRCODE 22
50. ISR\_NOERRCODE 23
51. ISR\_NOERRCODE 24
52. ISR\_NOERRCODE 25
53. ISR\_NOERRCODE 26
54. ISR\_NOERRCODE 27
55. ISR\_NOERRCODE 28
56. ISR\_NOERRCODE 29
57. ISR\_NOERRCODE 30
58. ISR\_NOERRCODE 31
59. ; 32～255 用户自定义
60. ISR\_NOERRCODE 255
61. [GLOBAL isr\_common\_stub]
62. [EXTERN isr\_handler]
63. ; 中断服务程序
64. isr\_common\_stub:
65. pusha ; Pushes edi, esi, ebp, esp, ebx, edx, ecx, eax
66. mov ax, ds
67. push eax ; 保存数据段描述符
69. mov ax, 0x10 ; 加载内核数据段描述符表
70. mov ds, ax
71. mov es, ax
72. mov fs, ax
73. mov gs, ax
74. mov ss, ax
76. push esp ; 此时的 esp 寄存器的值等价于 pt\_regs 结构体的指针
77. call isr\_handler ; 在 C 语言代码里
78. add esp, 4 ; 清除压入的参数
80. pop ebx ; 恢复原来的数据段描述符
81. mov ds, bx
82. mov es, bx
83. mov fs, bx
84. mov gs, bx
85. mov ss, bx
87. popa ; Pops edi, esi, ebp, esp, ebx, edx, ecx, eax
88. add esp, 8 ; 清理栈里的 error code 和 ISR
89. iret
90. .end:

在这段汇编代码中，我们首先定义了头文件里的加载idt 的函数。然后用宏来定义了中断处理函数的框架，后面29行到64行定义了void isr0()到void isr255()这些函数。再下面是函数公共处理的部分isr\_common\_stub，首先压入所有的通用寄存器的值，然后把ds压入，保存数据段描述符。后面的mov ax,0x10在上一章GDT那里也出现过，是因为我们的描述符前16位是段的长度，17位开始是基址。到了后面push esp的时候，可以回头去对照一下上面的pt\_regs的结构，看看是不是符合。

接下来的call isr\_handler就是针对各自的中断特定的操作了。操作之后，就是一些恢复工作。

最后看看我们的C语言部分函数的定义：

/idt/idt.c

1. #include "common.h"
2. #include "string.h"
3. #include "print.h"
4. #include "idt.h"
5. // 中断描述符表
6. i\_descriptor idt[256];
7. // IDTR
8. idtr idt\_ptr;
9. // 中断处理函数的指针数组
10. interrupt\_handler\_t interrupt\_handlers[256];
11. // 设置中断描述符
12. static void idt\_set\_gate(uint8\_t num, uint32\_t base, uint16\_t sel, uint8\_t flags);
13. // 声明加载 IDTR 的函数
14. extern void idt\_flush(uint32\_t);
15. //声明一个测试函数
16. static void interupt255\_test(pt\_regs \*regs);
17. // 初始化中断描述符表
18. void init\_idt()
19. {
20. bzero((uint8\_t \*)&interrupt\_handlers, sizeof(interrupt\_handler\_t) \* 256);
22. idt\_ptr.limit = sizeof(i\_descriptor) \* 256 - 1;
23. idt\_ptr.base = (uint32\_t)&idt;
25. bzero((uint8\_t \*)&idt, sizeof(i\_descriptor) \* 256);
26. // 0-32: 用于 CPU 的中断处理
27. idt\_set\_gate( 0, (uint32\_t)isr0, 0x08, 0x8E);
28. idt\_set\_gate( 1, (uint32\_t)isr1, 0x08, 0x8E);
29. idt\_set\_gate( 2, (uint32\_t)isr2, 0x08, 0x8E);
30. idt\_set\_gate( 3, (uint32\_t)isr3, 0x08, 0x8E);
31. idt\_set\_gate( 4, (uint32\_t)isr4, 0x08, 0x8E);
32. idt\_set\_gate( 5, (uint32\_t)isr5, 0x08, 0x8E);
33. idt\_set\_gate( 6, (uint32\_t)isr6, 0x08, 0x8E);
34. idt\_set\_gate( 7, (uint32\_t)isr7, 0x08, 0x8E);
35. idt\_set\_gate( 8, (uint32\_t)isr8, 0x08, 0x8E);
36. idt\_set\_gate( 9, (uint32\_t)isr9, 0x08, 0x8E);
37. idt\_set\_gate(10, (uint32\_t)isr10, 0x08, 0x8E);
38. idt\_set\_gate(11, (uint32\_t)isr11, 0x08, 0x8E);
39. idt\_set\_gate(12, (uint32\_t)isr12, 0x08, 0x8E);
40. idt\_set\_gate(13, (uint32\_t)isr13, 0x08, 0x8E);
41. idt\_set\_gate(14, (uint32\_t)isr14, 0x08, 0x8E);
42. idt\_set\_gate(15, (uint32\_t)isr15, 0x08, 0x8E);
43. idt\_set\_gate(16, (uint32\_t)isr16, 0x08, 0x8E);
44. idt\_set\_gate(17, (uint32\_t)isr17, 0x08, 0x8E);
45. idt\_set\_gate(18, (uint32\_t)isr18, 0x08, 0x8E);
46. idt\_set\_gate(19, (uint32\_t)isr19, 0x08, 0x8E);
47. idt\_set\_gate(20, (uint32\_t)isr20, 0x08, 0x8E);
48. idt\_set\_gate(21, (uint32\_t)isr21, 0x08, 0x8E);
49. idt\_set\_gate(22, (uint32\_t)isr22, 0x08, 0x8E);
50. idt\_set\_gate(23, (uint32\_t)isr23, 0x08, 0x8E);
51. idt\_set\_gate(24, (uint32\_t)isr24, 0x08, 0x8E);
52. idt\_set\_gate(25, (uint32\_t)isr25, 0x08, 0x8E);
53. idt\_set\_gate(26, (uint32\_t)isr26, 0x08, 0x8E);
54. idt\_set\_gate(27, (uint32\_t)isr27, 0x08, 0x8E);
55. idt\_set\_gate(28, (uint32\_t)isr28, 0x08, 0x8E);
56. idt\_set\_gate(29, (uint32\_t)isr29, 0x08, 0x8E);
57. idt\_set\_gate(30, (uint32\_t)isr30, 0x08, 0x8E);
58. idt\_set\_gate(31, (uint32\_t)isr31, 0x08, 0x8E);
59. // 255 将来用于实现系统调用
60. idt\_set\_gate(255, (uint32\_t)isr255, 0x08, 0x8E);
62. //注册测试函数到我们的中断中
63. register\_interrupt\_handler(255,interupt255\_test);
65. // 更新设置中断描述符表
66. idt\_flush((uint32\_t)&idt\_ptr);
67. }
68. // 设置中断描述符
69. static void idt\_set\_gate(uint8\_t num, uint32\_t base, uint16\_t sel, uint8\_t flags)
70. {
71. idt[num].base\_lo = base & 0xFFFF;
72. idt[num].base\_hi = (base >> 16) & 0xFFFF;
73. idt[num].sel = sel;
74. idt[num].always0 = 0;
75. // 先留下 0x60 这个魔数，以后实现用户态时候
76. // 这个与运算可以设置中断门的特权级别为 3
77. idt[num].flags = flags; // | 0x60
78. }
79. // 调用中断处理函数
80. void isr\_handler(pt\_regs \*regs)
81. {
82. if (interrupt\_handlers[regs->int\_no]) {
83. interrupt\_handlers[regs->int\_no](regs);
84. } else {
85. printk("Unhandled interrupt: %d\n", regs->int\_no);
86. }
87. }
88. // 注册一个中断处理函数
89. void register\_interrupt\_handler(uint8\_t n, interrupt\_handler\_t h)
90. {
91. interrupt\_handlers[n] = h;
92. }
93. //测试函数的具体内容
94. void interupt255\_test(pt\_regs \*regs){
95. printk("the interrupt test for IRQ\_255\n");
96. }

这部分和我们之前的gdt.c类似。首先7行处声明了idt，10行是idtr，16行是设置，19行是加载函数。注意22行这里我写了一个测试函数，待会儿我们中断的时候可以用这个。25行是初始化我们的中断描述符表，和gdt那套一样，不多说了。

这里我们特别看一下idt表里的参数，我们在idt\_set\_gate里段选择子设置了0x08这个值，这个值指的是在gdt或者ldt里面内核的代码段，我们上一章在gdt里设置了5个段，第一个是**NULL**段，第二个就是内核的数据段，而一个段描述符大小是8字节，所以我们这里偏移8字节，正好是我们的内核数据段。

后面0x8E就是flag部分，也就是39到46位，也就是1000 1110，我们来核对一下，39到43位01110代表了我们的中断是32位的**interrupt gate**。后面44到45位是00，也就是**ring 0**权限，最后46位的1表示段存在。

再下面在72行，我们把我们上面的测试函数注册到我们的中断处理函数中，最后就是idt\_flush((uint32\_t)&idt\_ptr)，加载我们的idt，这个函数的实现在汇编里。

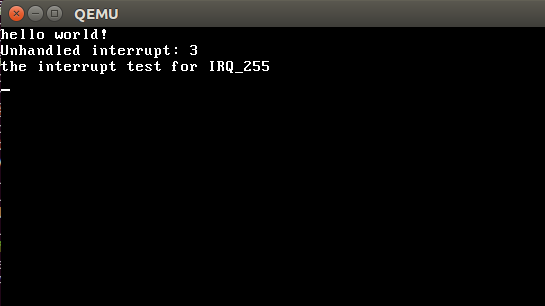
最后我们捋一下，如果我们写一个中断，比如int $0x9，会发生什么呢：

好了终于把所有内容都解释完了。我们在entry.c里面写一下我们的中断，看看会发生什么：

/entry.c

1. #include"console.h"
2. #include"print.h"
3. #include"gdt.h"
4. #include"idt.h"
5. int kern\_entry()
6. {
7. init\_gdt();
8. init\_idt();
9. console\_clear();
10. printk("hello world!\n");
11. asm volatile ("int $0x3");
12. asm volatile ("int $0xFF");
13. return 0;
14. }

执行结果如下：



我们一共调用了2个中断，一个3号中断，我们没有说处理方式，于是输出unhandled interrupt，我们自己注册了一个255号中断，输出了我们的内容。

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

├── idt

│   ├── idt.c

│   └── idt\_s.s

├── include

│   ├── common.h

│   ├── console.h

│   ├── gdt.h

│   ├── idt.h

│   ├── print.h

│   ├── string.h

│   ├── types.h

│   └── vargs.h

├── kernel

│   └── print.c

├── libs

│   ├── common.c

│   └── string.c

├── Makefile

└── scripts

├── gdbinit

└── kernel.ld

8 directories, 22 files

<http://www.brokenthorn.com/Resources/OSDev15.html>

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

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

<https://en.wikipedia.org/wiki/Intel_8259>

<https://zh.wikipedia.org/wiki/Intel_8259>

<http://wiki.osdev.org/8259_PIC>

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

https://en.wikipedia.org/wiki/Interrupt\_vector\_table

<http://wiki.osdev.org/IDT>

https://msdn.microsoft.com/en-us/library/ms892408.aspx