

练习 6：完善中断初始化和处理（需要编程）

请完成编码工作并回答如下问题。

(1) 中断向量表中一个表项占多少个字节？其中哪几位代表中断处理代码的入口？

(2) 请编程完善 `kern/trap/trap.c` 中对中断向量表进行初始化的函数 `idt_init`。在 `idt_init` 函数中，依次对所有中断入口进行初始化。使用 `mmu.h` 中的 `SETGATE` 宏，填充 `idt` 数组内容。注意除了系统调用中断（`T_SYSCALL`）以外，其他中断均使用中断门描述符，权限为内核态权限；而系统调用中断使用异常，权限为陷阱门描述符。每个中断的入口由 `tools/vectors.c` 生成，使用 `trap.c` 中声明的 `vectors` 数组即可。

(3) 请编程完善 `trap.c` 中的中断处理函数 `trap`，在对时钟中断进行处理的部分，请填写 `trap` 函数中处理时钟中断的部分，使操作系统每遇到 100 次时钟中断后，调用 `print_ticks` 子程序，向屏幕上打印一行文字“100 ticks”。

要求完成问题（2）和问题（3）提出的相关函数实现，提交改进后的源代码包（可以编译执行），并在实验报告中简要说明实现过程，并写出对问题 1 的回答。完成问题（2）和问题（3）要求的部分代码后，运行整个系统，可以看到大约每 1s 会输出一行“100 ticks”，而按下的键也会在屏幕上显示。

提示：可阅读 2.3.3 节中的“中断与异常”。

答：

(1) 在文件 `kern/mm/mmu.h` 中，中断向量表的结构定义如下：

```
49  /* 中断门和陷阱门描述符 */
50  struct gatedesc {
51      unsigned gd_off_15_0 : 16;          // 偏移量的低 16 位
52      unsigned gd_ss : 16;                 // 段选择子
53      unsigned gd_args : 5;               // # args, 0 for
      interrupt/trap gates
54      unsigned gd_rsv1 : 3;                // reserved(should
      be zero I guess)
55      unsigned gd_type : 4;               //
      type(STS_{TG,IG32,TG32})
56      unsigned gd_s : 1;                  // must be 0 (system)
57      unsigned gd_dpl : 2;               // descriptor(meaning
      new) privilege level
58      unsigned gd_p : 1;                  // Present
59      unsigned gd_off_31_16 : 16;        // 偏移量的高 16 位
60  };
```

可以看到代码中使用位域来定义，一共占了 64 个位，即 8 个字节，其中 0-1 字节和 6-7 字节分别是偏移量低 16 位和高 16 位，代表了中断处理代码的入口，2-3 字节是段选择子，如下表所示：

表 1 IDT 入口，中断门^[1]

名称	位	全称	说明
Offset	48..63	Offset 16..31	偏移量的高 12 位
P	47	Present	未使用的中断应置为 0
DPL	45,46	Descriptor Privilege Level	门调用保护。指定调用描述符应具有的限制级别的最小值。因此可以保护硬件和 CPU 中断不被用户空间调用。
S	44	Storage Segment	中断门和陷阱门应置为 0
Type	40..43	Gate Type 0..3	可能的 IDT 门类型有： 0101b 32 bit task gate 0110b 16-bit interrupt gate 0111b 16-bit trap gate 1110b 32-bit interrupt gate 1111b 32-bit trap gate
0	32..39	Unused 0..7	保留，应置为 0
Selector	16..31	Selector 0..15	中断函数的选择器（内核的选择器才有意义）。选择器的描述符的 DPL 字段必须为 0，因此 iret 指令在执行时不会抛出 #GP 异常。
Offset	0..15	Offset 0..15	中断函数偏移地址的低 16 位

当 Intel CPU 运行在 32 位保护模式下时，需要使用中断描述符表（Interrupt Descriptor Table, IDT）来管理中断或异常。IDT 是 Intel 8086~80186 CPU 中使用的中断向量表的直接替代物。其作用也类似于中断向量表，只是其中每个中断描述符项中除了含有中断服务程序地址以外，还包含有关特权级和描述符类别等信息。^[2]

在保护模式下，中断描述符表中的每个表项由 8 个字节组成，其中的每个表项叫做一个门描述符（Gate Descriptor），“门”的含义是指当中断发生时必须先访问这些“门”，能够“开门”（即将要进行的处理需通过特权检查，符合设定的权限等约束）后，然后才能进入相应的处理程序。而门描述符则描述了“门”的属性（如特权级、段内偏移量等）。

在 IDT 中，可以包含如下 3 种类型的系统段描述符：

- **中断门描述符（Interrupt-gate descriptor）**：用于中断处理，其类型码为 110，中断门包含了一个外设中断或故障中断的处理程序所在段的选择子和段内偏移量。当控制权通

[1] <https://blog.csdn.net/chen1540524015/article/details/73817554>

[2] <https://www.xuebuyuan.com/1160926.html>

过中断门进入中断处理程序时，处理器清 IF 标志，即关中断，以避免嵌套中断的发生。中断门中的 DPL（Descriptor Privilege Level）为 0，因此用户态的进程不能访问中断门。所有的中断处理程序都由中断门激活，并全部限制在内核态。

- **陷阱门描述符（Trap-gate descriptor）**：用于系统调用，其类型码为 111，与中断门类似，其唯一的区别是，控制权通过陷阱门进入处理程序时维持 IF 标志位不变，也就是说，不关中断。
- **任务门描述符（Task-gate descriptor）和调用门描述符（Call-gate descriptor）**：这两种主要是 Intel 设置的“任务”切换的手段。^[3]

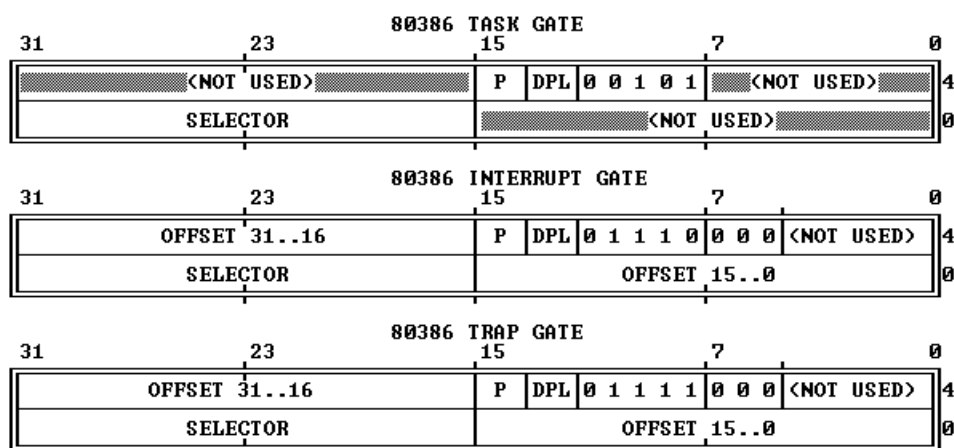


图 1 80386 的 IDT 门描述符

(2) 函数 `idt_init` 代码如下^[4]:

```
/* idt_init - 为 kern/trap/vectors.S 中每一个入口点初始化 IDT */
void
idt_init(void) {
    // 保存在 vectors.S 中的 256 个中断处理例程的入口地址数组
    extern uintptr_t __vectors[];
    int i;
    // 在中断门描述符表中通过建立中断门描述符，其中存储了中断处理例程的代码段 GD_KTEXT 和偏移量 __vectors[i]，特权级为 DPL_KERNEL。
    // 这样通过查询 idt[i] 就可定位到中断服务例程的起始地址
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i],
DPL_KERNEL);
    }
    // 设置用户态到内核态的转换
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT,
```

[3] https://wiki.osdev.org/Interrupt_Descriptor_Table

[4] https://www.bookstack.cn/read/simple_os_book/zh-chapter-2-init_IDT.md

```
__vectors[T_SWITCH_TOK], DPL_USER);
    // 建立好中断门描述符表后,通过指令 lidt 把中断门描述符表的起始
    地址装入 IDTR 寄存器中,从而完成中段描述符表的初始化工作
    lidt(&idt_pd);
}
```

`__vectors` 数组中存放了 `vectors.S` 中的 256 个中断处理例程的入口地址, `vectors.S` 是汇编语言文件, 它由工具 `tools/vector.c` 执行得到。

第 (1) 题列出了 80386 中断描述符表中每个表项 (中断门描述符和陷阱门描述符) 的各个位的含义, `idt_init` 函数就负责将这些表项建立好。中断描述符表是 `struct gatedesc` 类型的数组, 它由最多 256 个中断描述符构成, 每个索引 (0-255) 代表一个中断号。

当一个中断产生之后, 它会有对应的中断号, 这个中断号用来作为索引在 `IDT` 描述符表中进行查询, 每个表项记录了中断服务例程的地址, 它包含段描述符 (段选择子) 和偏移量, 可以根据段选择子的内容, 查找全局描述符表 (`GDT`) 中的段描述符, 然后就能得到中断服务例程的基址, 与偏移量 `offset` 相加就可以得到中断服务例程的起始地址, 整个查找的过程是由硬件完成的。

全局描述符表和中断描述符表都是由软件来建立的, 这里使用 `lidt` 指令完成对中段描述符表的加载, 使 CPU 知道中断描述符表在内存中所处的位置。

(3) 要让 x86 的中断系统能够正常工作, 首先需要做几件事情, 见 `kern/init/init.c` 中的如下代码:

```
32     pic_init();                // 中断控制器的初始化
33     idt_init();                // 中断描述符表的初始化
34
35     clock_init();              // 时钟中断的初始化
36     intr_enable();            // 使能中断
```

8259 中断控制器是一个外设, 它是一个特殊的设备, `pic_init` 函数完成对 8259 的管理和配置, 只有将 8259 配置好之后, 相应的外设才能产生中断并被 CPU 所接收和处理, 为接受后续的时钟中断做铺垫。

中断控制器初始化之后, 需要让 CPU 建立一个中断描述符表 (`IDT`) 并进行初始化, 具体过程参见第 (2) 题。如果某一个外设想要能够产生中断, 需要对特定的外设也做相应的初始化, 所以如果我们这里面需要处理时钟中断, 就需要使用 `clock_init` 函数对时钟中断进行初始化。`BootLoader` 启动的时候是处于屏蔽中断的状态, 因此最后还要使用 `intr_enable` 函数进行使能中断。

`clock_init` 函数位于 `kern/driver/clock.c` 中, 其代码如下:

```

28  /* *
29  * clock_init - initialize 8253 clock to interrupt 100
    times per second,
30  * and then enable IRQ_TIMER.
31  * */
32  void
33  clock_init(void) {
34      // set 8253 timer-chip
35      outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN |
    TIMER_16BIT);
36      outb(IO_TIMER1, TIMER_DIV(100) % 256);
37      outb(IO_TIMER1, TIMER_DIV(100) / 256);
38
39      // initialize time counter 'ticks' to zero
40      ticks = 0;
41
42      cprintf("++ setup timer interrupts\n");
43      pic_enable(IRQ_TIMER);
44  }

```

这里面包含了跟时钟外设 8253 芯片相关的一些设置，作用是每 100 个 tick 会产生一次中断。

kern/driver/intr.c 中的代码如下：

```

4  /* intr_enable - enable irq interrupt */
5  void
6  intr_enable(void) {
7      sti();
8  }
9
10 /* intr_disable - disable irq interrupt */
11 void
12 intr_disable(void) {
13     cli();
14 }

```

从中可以看到，使能中断就是机器指令 STI，而 CLI 是另一条屏蔽中断的指令。

CPU 在接收到中断信号之后要做的事情：

1. 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中断控制器（如：8259A）是否发送中断请求过来，如果有那么 CPU 就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量；
2. CPU 根据得到的中断向量（以此为索引）到 IDT 中找到该向量对应的中断描述符，中断描述符里保存着中断服务例程的段选择子；
3. CPU 使用 IDT 查到的中断服务例程的段选择子从 GDT 中取得相应的段描述符，段描述符里保存了中断服务例程的段基址和属性信息，此时 CPU 就得到了中断服务例程的起始地址，并跳转到该地址；
4. CPU 会根据 CPL 和中断服务例程的段描述符的 DPL 信息确认是否发生了特权级的转换。比如当前程序正运行在用户态，而中断程序是运行在内核态的，则意味着发生了特权级的转换，这时 CPU 会从当前程序的 TSS 信息（该信息在内存中的起始地址存在 TR 寄存器中）里取得该程序的内核栈地址，即包括内核态的 SS 和 ESP 的值，并立即将系统当前使用的栈切换成新的内核栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的用户态的 SS 和 ESP 压到新的内核栈中保存起来；
5. CPU 需要开始保存当前被打断的程序的现场（即一些寄存器的值），以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息，即依次压入当前被打断程序使用的 Eflags, CS, EIP, Error Code（如果有错误码的异常）信息；
6. CPU 利用中断服务例程的段描述符将其第一条指令的地址加载到 CS 和 EIP 寄存器中，开始执行中断服务例程。这意味着先前的程序被暂停执行，中断服务程序正式开始工作。

前面 `trap/vector.S` 中定义了 255 个中断号所对应的起始地址，可以看到每个中断服务例程都通过 `jmp __alltraps` 跳到 `trap/trapentry.S` 中的 `__alltraps` 这个入口，`__alltraps` 会保存一系列的寄存器。硬件产生中断之后会保存被打断的地址和它的 `flag` 寄存器等等，但是它保存并不完整，那么为了能够回到被打断的地方重新执行呢？我们需要确保环境能够完全恢复到跟以前一样，所以我们把后续在执行中断服务例程中用到的寄存器的内容都要先保存起来，以避免破坏返回去的那个环境。其中第 4、5 步一个重要的功能就是向内核栈中压入各种寄存器，压入这些寄存器既可以起到保存现场的作用，又能让中断服务例程（ISR）知道中断的各种信息。

可以看到 `trap/trapentry.S` 中保存寄存器放入内核的中断栈的代码如下：

```
9      pushl %ds
10     pushl %es
11     pushl %fs
12     pushl %gs
13     pushal
```

此后第 24 行的 `call trap` 实际上调用了函数 `trap`，它位于 `trap/trap.c` 中，`trap` 函数会继续调用 `trap_dispatch` 函数，进而查找这个相应中断号，如果说它发现是时钟中断，那么 `case IRQ_OFFSET + IRQ_TIMER` 就会成立，进而执行如下我们书写的代码，其作用就是每 `TICK_NUM`（这里是 100）个 `ticks` 通过 `print_ticks` 函数输出一次内容：

```

case IRQ_OFFSET + IRQ_TIMER:
    ticks++;
    if (ticks % TICK_NUM == 0) {
        print_ticks();
    }
    break;

```

print_ticks 函数的内容如下:

```

14 static void print_ticks() {
15     cprintf("%d ticks\n", TICK_NUM);
16 #ifdef DEBUG_GRADE
17     cprintf("End of Test.\n");
18     panic("EOT: kernel seems ok.");
19 #endif
20 }

```

在被打断的那一刻,除了 CPU 要压入的各种寄存器,我们还需要压入其他一些寄存器用于保存现场和提供给 ISR 中断信息。在 uCore 中,我们使用结构体 trapframe 来将保存的寄存器传给 ISR。

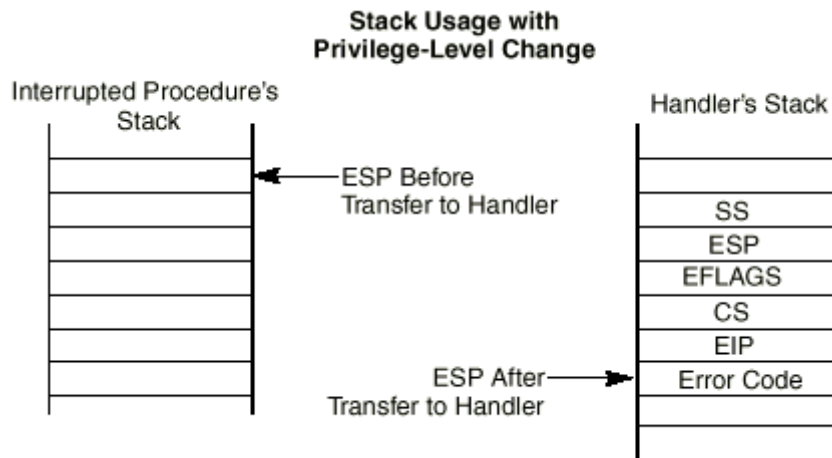


图 2 特权级转换后栈空间变化示意图

trap/trap.h 中结构体 trapframe 的定义如下:

```

50 /* registers as pushed by pushal */
51 struct pushregs {
52     uint32_t reg_edi;
53     uint32_t reg_esi;
54     uint32_t reg_ebp;
55     uint32_t reg_oesp; /* Useless */

```

```

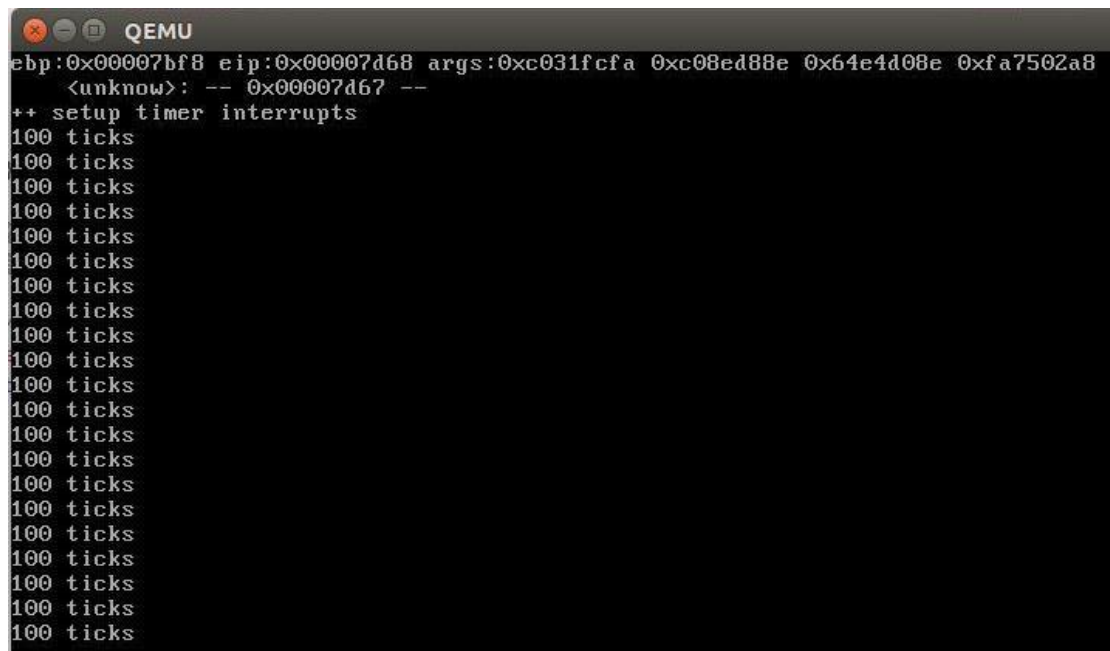
56     uint32_t reg_ebx;
57     uint32_t reg_edx;
58     uint32_t reg_ecx;
59     uint32_t reg_eax;
60 };
61
62 struct trapframe {
63     struct pushregs tf_regs;
64     uint16_t tf_gs;
65     uint16_t tf_padding0;
66     uint16_t tf_fs;
67     uint16_t tf_padding1;
68     uint16_t tf_es;
69     uint16_t tf_padding2;
70     uint16_t tf_ds;
71     uint16_t tf_padding3;
72     uint32_t tf_trapno;
73     /* below here defined by x86 hardware */
74     uint32_t tf_err;
75     uintptr_t tf_eip;
76     uint16_t tf_cs;
77     uint16_t tf_padding4;
78     uint32_t tf_eflags;
79     /* below here only when crossing rings, such as from
        user to kernel */
80     uintptr_t tf_esp;
81     uint16_t tf_ss;
82     uint16_t tf_padding5;
83 } __attribute__((packed));

```

其中第 74-78 行是在硬件产生中断之后硬件 CPU 自动保存的一些信息；第 63-72 行是软件保存的信息，pushregs 中的寄存器都是 pushal 中需要压入栈的所有寄存器；第 80-82 行考虑的是将来有可能出现从用户态产生中断会切换到内核态，那么就会多保存一些信息如用户态的栈（ESP、SS）。对于 x86 而言，用户态一般我们设置在特权级 3，而内核态设置在特权级 0。有了这个数据结构后，我们就可以在中断后获取中断的信息，并将它传给 ISR，ISR 会根据传入的 trapframe 来进行相应的操作。^[5]

最终执行的效果如下：

[5] <https://www.jianshu.com/p/94fec16c5252>



A screenshot of a QEMU terminal window. The title bar shows the QEMU logo and the text "QEMU". The terminal displays assembly code and its execution output. The assembly code includes a function call to "setup timer interrupts" and a series of "100 ticks" instructions. The output shows the execution of these instructions, with the "100 ticks" instruction being repeated 16 times.

```
ebp:0x00007bf8 eip:0x00007d68 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknown>: -- 0x00007d67 --
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

图 3 执行结果