练习 5：实现函数调用堆栈跟踪函数

print_stackframe 函数源码如下：

```
void print_stackframe(void)
{
    //首先定义两个局部变量 ebp、esp 分别存放 ebp、esp 寄存器的值。这里将 ebp 定义为指针，是为了方便后面取 ebp 寄存器的值。
    uint32_t esp=0;
    uint32_t *ebp=0;
    /*调用 read_ebp 函数来获取执行 print_stackframe 函数时 ebp 寄存器的值，
    这里 read_ebp 必须定义为 inline 函数，否则获取的是执行 read_ebp 函数时的 ebp 寄存器的值。
    调用 read_eip 函数来获取当前指令的位置，也就是此时 eip 寄存器的值。
    这里 read_eip 必须定义为常规函数而不是 inline 函数，
    因为这样的话在调用 read_eip 时会把当前指令的下一条指令的地址（也就是 eip 寄存器的值）压栈，
    那么在进入 read_eip 函数内部后便可以从栈中获取到调用前 eip 寄存器的值。*/
    esp=read_eip();
    ebp=read_ebp();
    while(ebp)
    {
        cprintf("ebp:0x%08x eip:0x%08x args:", (uint32_t)ebp, esp);
        cprintf("0x%08x 0x%08x 0x%08x 0x%08x\n", ebp[2], ebp[3], ebp[4], ebp[5]);
     /*由于变量 eip 存放的是下一条指令的地址，因此将变量 eip 的值减去 1，
       得到的指令地址就属于当前指令的范围了。
       由于只要输入的地址属于当前指令的起始和结束位置之间，
       print_debuginfo 都能搜索到当前指令，因此这里减去 1 即可。
     */
        print_debuginfo(esp - 1);
    /*以后变量 eip 的值就不能再调用 read_eip 来获取了（每次调用获取的值都是相同的），
      而应该从 ebp 寄存器指向栈中的位置再往上一个单位中获取。
      由于 ebp 寄存器指向栈中的位置存放的是调用者的 ebp 寄存器的值，
      据此可以继续顺藤摸瓜，不断回溯，直到 ebp 寄存器的值变为 0
    */
        esp = ebp[1];
        ebp = (uint32_t *)*ebp;
    }
}
```

运行结果如下：

```
Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x00103780 (phys)
  edata  0x0010e950 (phys)
  end    0x0010fdc0 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b38 eip:0x00100bd0 args:0x00010094 0x0010e950 0x00007b68 0x001000a2
    kern/debug/kdebug.c:0: print_stackframe+37
ebp:0x00007b48 eip:0x00100f29 args:0x00000000 0x00000000 0x00000000 0x0010008d
    kern/debug/kmonitor.c:125: mon_backtrace+23
ebp:0x00007b68 eip:0x001000a2 args:0x00000000 0x00007b90 0xffff0000 0x00007b94
    kern/init/init.c:48: grade_backtrace2+32
ebp:0x00007b88 eip:0x001000d1 args:0x00000000 0xffff0000 0x00007bb4 0x001000e5
    kern/init/init.c:53: grade_backtrace1+37
ebp:0x00007ba8 eip:0x001000f8 args:0x00000000 0x00100000 0xffff0000 0x00100109
    kern/init/init.c:58: grade_backtrace0+29
ebp:0x00007bc8 eip:0x00100124 args:0x00000000 0x00000000 0x00000000 0x00103780
    kern/init/init.c:63: grade_backtrace+37
ebp:0x00007be8 eip:0x00100066 args:0x00000000 0x00000000 0x00000000 0x00007c4f
    kern/init/init.c:28: kern_init+101
ebp:0x00007bf8 eip:0x00007d6e args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d6d --
```

练习 6：完善中断初始化和处理

1. 中断描述符表中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

中断描述符表一个表项占 8 个字节，其结构如下：
bit 63-48：　offset 31-16
bit 47-32：　属性信息，包括 DPL、Pflag 等
bit 31-16：　Segment selector
bit 15-0：　offset 15-0
其中第 16-32 位是段选择子，用于索引全局描述符表 GDT 来获取中断处理代码对应的段地址，再加上第 0-15、48-63 位构成的偏移地址，即可得到中断处理代码的入口。

2. 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

中断处理函数的段选择子及偏移量的设置要参考 kern/trap/vectors.S 文件：由该文件可知，所有中断向量的中断处理函数地址均保存在__vectors 数组中，该数组中第 i 个元素对应第 i 个中断向量的中断处理函数地址。而且由文件开头可知，中断处理函数属于.text 的内容。因此，中断处理函数的段选择子即.text 的段选择子 GD_KTEXT。从 kern/mm/pmm.c 可知.text

的段基址为 0，因此中断处理函数地址的偏移量等于其地址本身。

```
/* idt_init - initialize IDT to each of the entry points in
kern/trap/vectors.S */
void idt_init(void) {
    extern uintptr_t __vectors[];
    uint32_t pos;

    for (pos = 0; pos < 256; pos++)
    {
        SETGATE(idt[pos], 0, GD_KTEXT, __vectors[pos], 0);
    }

    SETGATE(idt[T_SYSCALL], 1, sel, __vectors[T_SYSCALL], 3);
    lidt(&idt_pd);
}
```

3. 编程完善 trap.c 中的中断处理函数 trap

trap 函数只是直接调用了 trap_dispatch 函数，而 trap_dispatch 函数实现对各种中断的处理，题目要求我们完成对时钟中断的处理，实现非常简单：定义一个全局变量 ticks，每次时钟中断将 ticks 加 1，加到 100 后打印"100 ticks"，然后将 ticks 清零重新计数。代码实现如下：

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