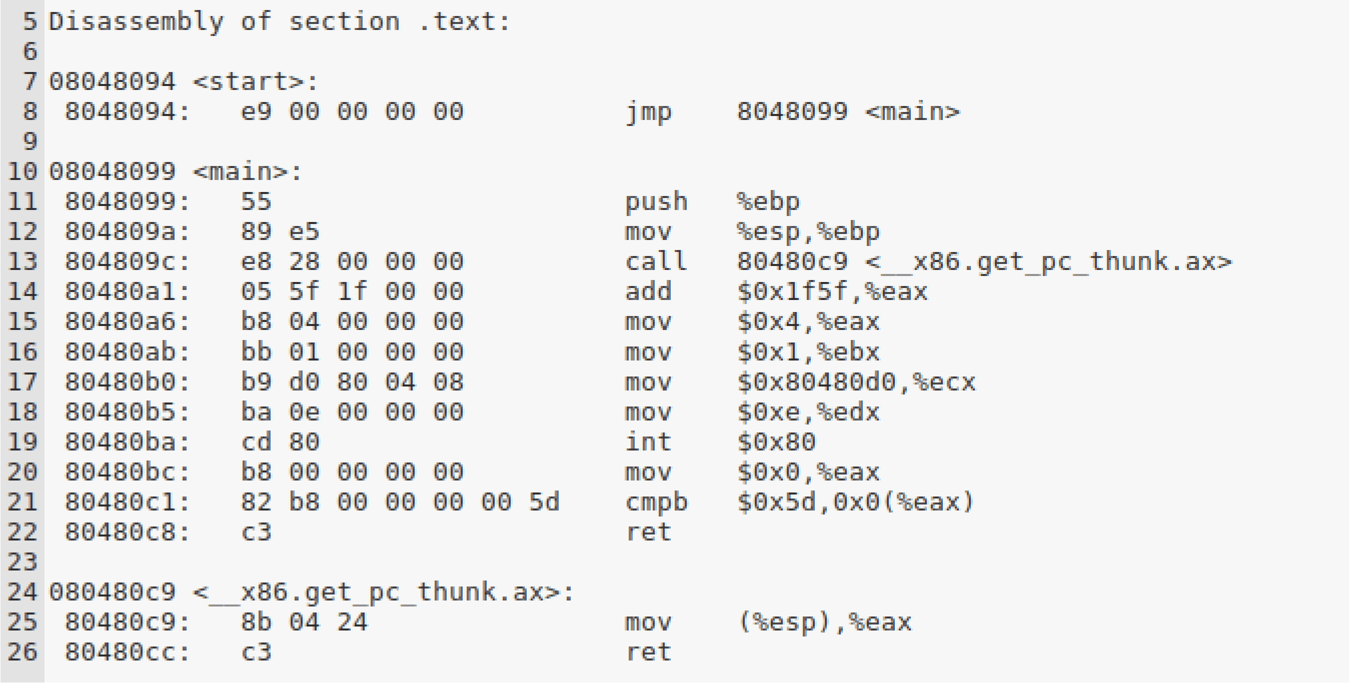
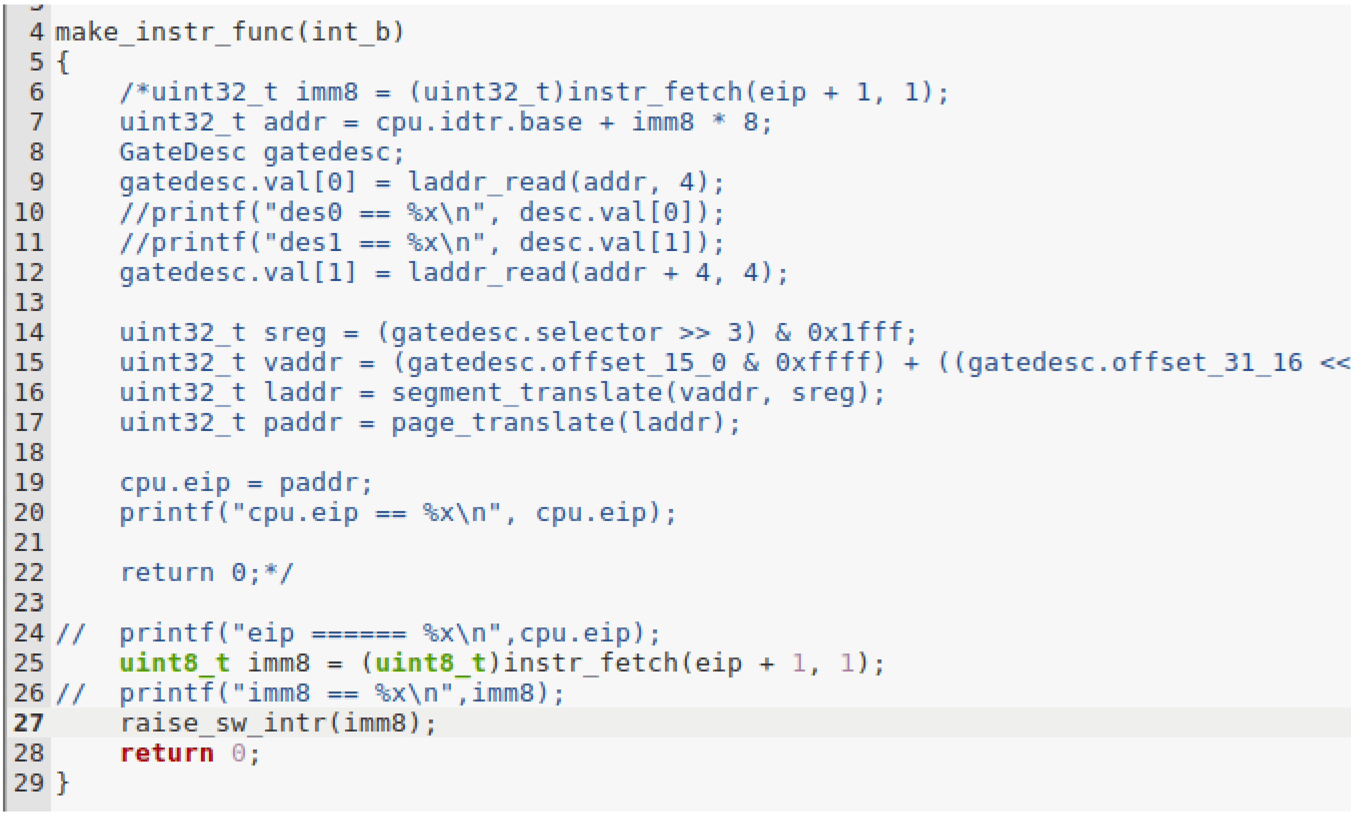
实验进度：完成了所有必需的内容，并且可以玩打字小游戏和仙剑

1. 详细描述从测试用例中的 int $0x80 开始一直到 HIT_GOOD_TRAP 为止的详细的系统行为（完整描述控 制的转移过程，即相关函数的调用和关键参数传递过程），可以通过文字或画图的方式来完成；



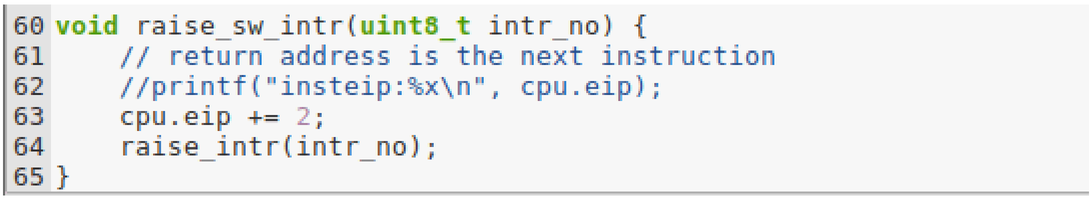
```
Disassembly of section .text:

08048094 <start>:
 8048094:   e9 00 00 00 00          jmp    8048099 <main>

08048099 <main>:
 8048099:   55                      push   %ebp
 804809a:   89 e5                   mov    %esp,%ebp
 804809c:   e8 28 00 00 00          call   80480c9 <__x86.get_pc_thunk.ax>
 80480a1:   05 5f 1f 00 00          add    $0x1f5f,%eax
 80480a6:   b8 04 00 00 00          mov    $0x4,%eax
 80480ab:   bb 01 00 00 00          mov    $0x1,%ebx
 80480b0:   b9 d0 80 04 08          mov    $0x80480d0,%ecx
 80480b5:   ba 0e 00 00 00          mov    $0xe,%edx
 80480ba:   cd 80                   int    $0x80
 80480bc:   b8 00 00 00 00          mov    $0x0,%eax
 80480c1:   82 b8 00 00 00 00 5d    cmpb   $0x5d,0x0(%eax)
 80480c8:   c3                      ret

080480c9 <__x86.get_pc_thunk.ax>:
 80480c9:   8b 04 24                mov    (%esp),%eax
 80480cc:   c3                      ret
```

这是 hello-inline 的测试用例的反汇编代码，可以看出在 0x80480ba 的地方进入了一个系统中断 int 0x80，这个时候调用 int 指令进行响应。



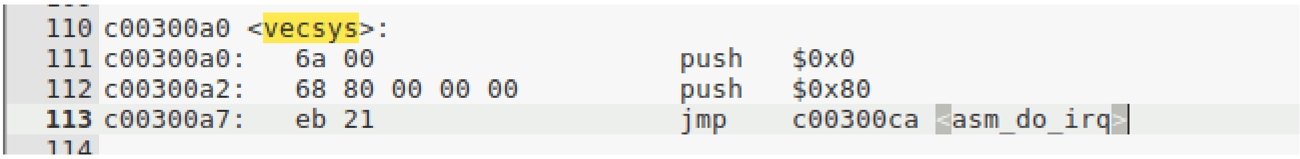
```
make_instr_func(int_b)
{
    /*uint32_t imm8 = (uint32_t)instr_fetch(eip + 1, 1);
    uint32_t addr = cpu.idtr.base + imm8 * 8;
    GateDesc gatedesc;
    gatedesc.val[0] = laddr_read(addr, 4);
    //printf("des0 == %x\n", desc.val[0]);
    //printf("des1 == %x\n", desc.val[1]);
    gatedesc.val[1] = laddr_read(addr + 4, 4);

    uint32_t sreg = (gatedesc.selector >> 3) & 0x1fff;
    uint32_t vaddr = (gatedesc.offset_15_0 & 0xffff) + ((gatedesc.offset_31_16 <<
    uint32_t laddr = segment_translate(vaddr, sreg);
    uint32_t paddr = page_translate(laddr);

    cpu.eip = paddr;
    printf("cpu.eip == %x\n", cpu.eip);

    return 0;*/

//  printf("eip ====== %x\n",cpu.eip);
    uint8_t imm8 = (uint8_t)instr_fetch(eip + 1, 1);
//  printf("imm8 == %x\n",imm8);
    raise_sw_intr(imm8);
    return 0;
}
```

在 int 中，用 instr_fetch 把中断号 0x80 传给函数 raise_sw_instr 函数。



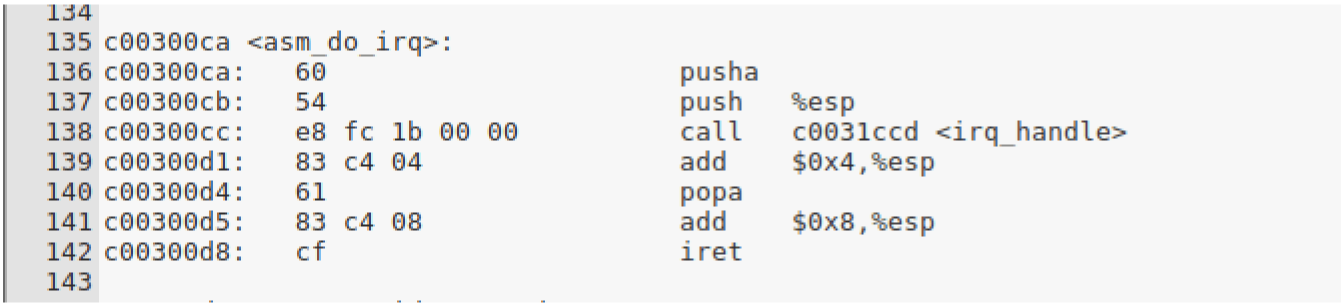
```
void raise_sw_intr(uint8_t intr_no) {
    // return address is the next instruction
    //printf("insteip:%x\n", cpu.eip);
    cpu.eip += 2;
    raise_intr(intr_no);
}
```

这个时候看到了 eip+=2，因为 int 0x80 是 2 个字节的指令码，所以这时候的 eip 就指向了在 hello-inline 用例里 int 0x80 的下一条指令。这时候调用 raise_intr。raise_intr 里面保存了 eflags　cs　eip 的值，然后进入内核态，由系统处理中断。



```
c00300a0 <vecsys>:
c00300a0:	6a 00                	push   $0x0
c00300a2:	68 80 00 00 00       	push   $0x80
c00300a7:	eb 21                	jmp    c00300ca <asm_do_irq>
```

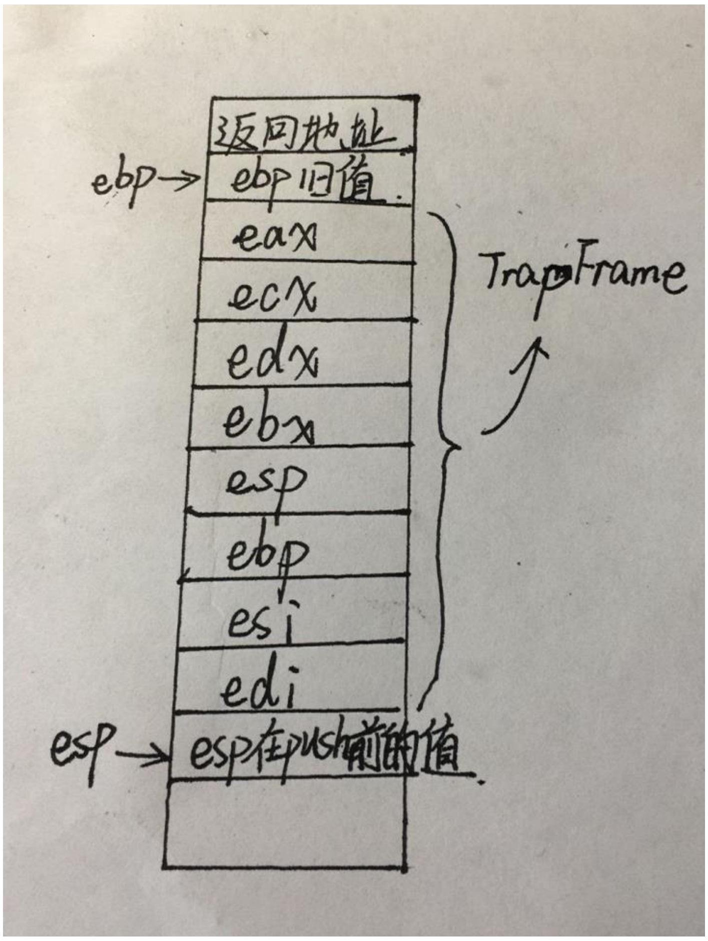
然后进入 vecsys 这个调用，调用响应的处理函数 asm_do_irq。



```
c00300ca <asm_do_irq>:
c00300ca:	60                   	pusha  
c00300cb:	54                   	push   %esp
c00300cc:	e8 fc 1b 00 00       	call   c0031ccd <irq_handle>
c00300d1:	83 c4 04             	add    $0x4,%esp
c00300d4:	61                   	popa   
c00300d5:	83 c4 08             	add    $0x8,%esp
c00300d8:	cf                   	iret   
```

pusha 保存之前寄存器的值，然后调用 irq_handle 进行具体的处理，最后 popa 还原之前的寄存器的值，最后 iret 将 eip cs eflags 的现场还原出来，这时候将控制权转移进用户态，eip 还原出来的值就是在 hello-inline 用例里 int 0x80 的下一条指令，这时用户进程继续执行，最后 HIT_GOOD_TRAP。

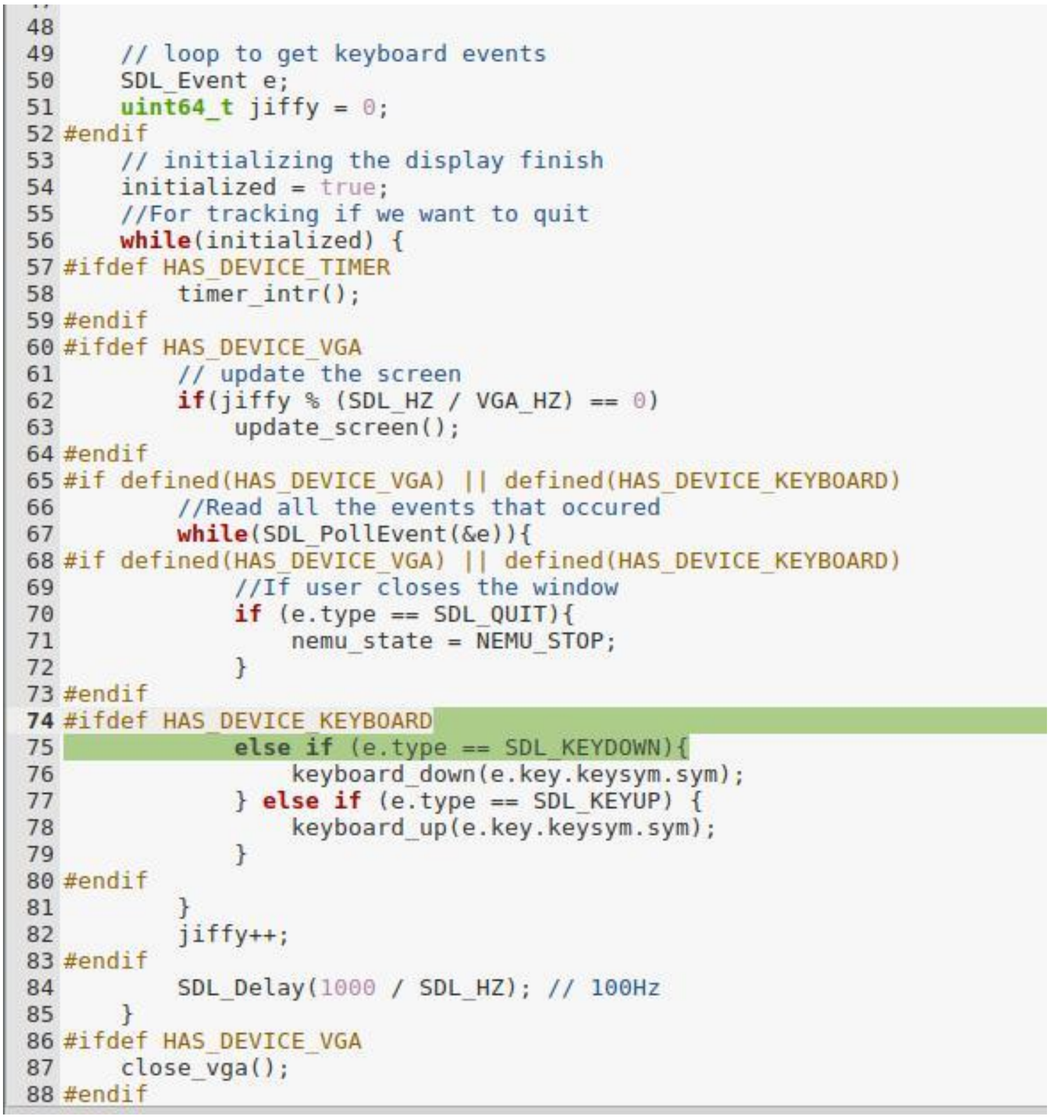
2、在描述过程中，回答 kernel/src/irq/do_irq.S 中的 push %esp 起什么作用，画出在 call irq_handle 之前，系统栈的内容和 esp 的位置，指出 TrapFrame 对应系统栈的哪一段内容。



3、详细描述 NEMU 和 Kernel 响应时钟中断的过程和先前的系统调用过程不同之处在哪里？相同的地方 又在哪里？可以通过文字或画图的方式来完成

相同的地方就是系统在发生中断后，都会保存现场，然后将控制权转移到内核态去处理，处理完毕后恢复现场，将控制权转移给用户态继续执行进程。
不同的地方就是，先前的系统调用是通过 int 指令来实现的，而时钟中断是每次执行完一条指令后调用 do_intr()函数查看并处理中断事件，属于系统实时监测中断的产生再做相应的处理。

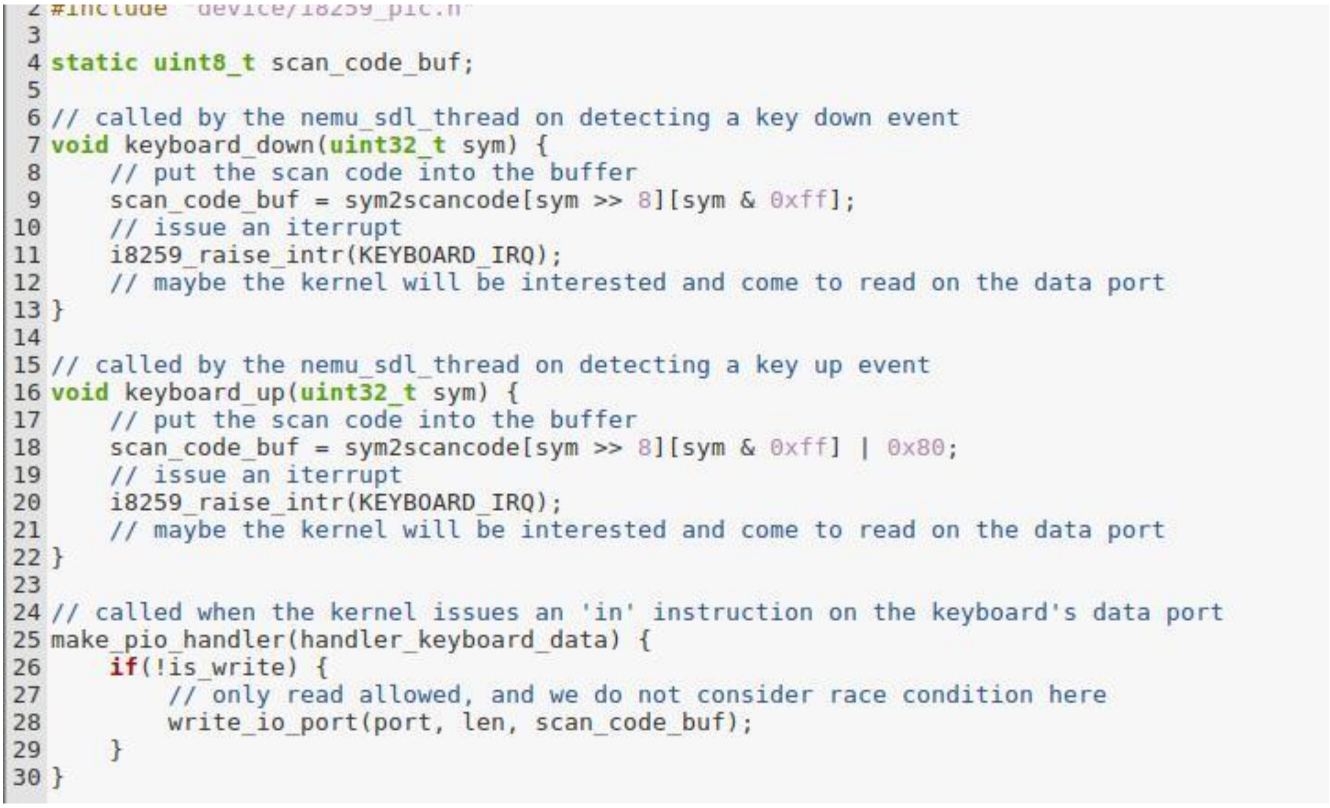
4. 注册监听键盘事件是怎么完成的？



```

    // loop to get keyboard events
    SDL_Event e;
    uint64_t jiffy = 0;
#endif
    // initializing the display finish
    initialized = true;
    //For tracking if we want to quit
    while(initialized) {
#ifdef HAS_DEVICE_TIMER
        timer_intr();
#endif
#ifdef HAS_DEVICE_VGA
        // update the screen
        if(jiffy % (SDL_HZ / VGA_HZ) == 0)
            update_screen();
#endif
#if defined(HAS_DEVICE_VGA) || defined(HAS_DEVICE_KEYBOARD)
        //Read all the events that occured
        while(SDL_PollEvent(&e)){
#if defined(HAS_DEVICE_VGA) || defined(HAS_DEVICE_KEYBOARD)
            //If user closes the window
            if (e.type == SDL_QUIT){
                nemu_state = NEMU_STOP;
            }
#endif
#ifdef HAS_DEVICE_KEYBOARD
            else if (e.type == SDL_KEYDOWN){
                keyboard_down(e.key.keysym.sym);
            } else if (e.type == SDL_KEYUP) {
                keyboard_up(e.key.keysym.sym);
            }
#endif
        }
        jiffy++;
#endif
        SDL_Delay(1000 / SDL_HZ); // 100Hz
    }
#ifdef HAS_DEVICE_VGA
    close_vga();
#endif
```

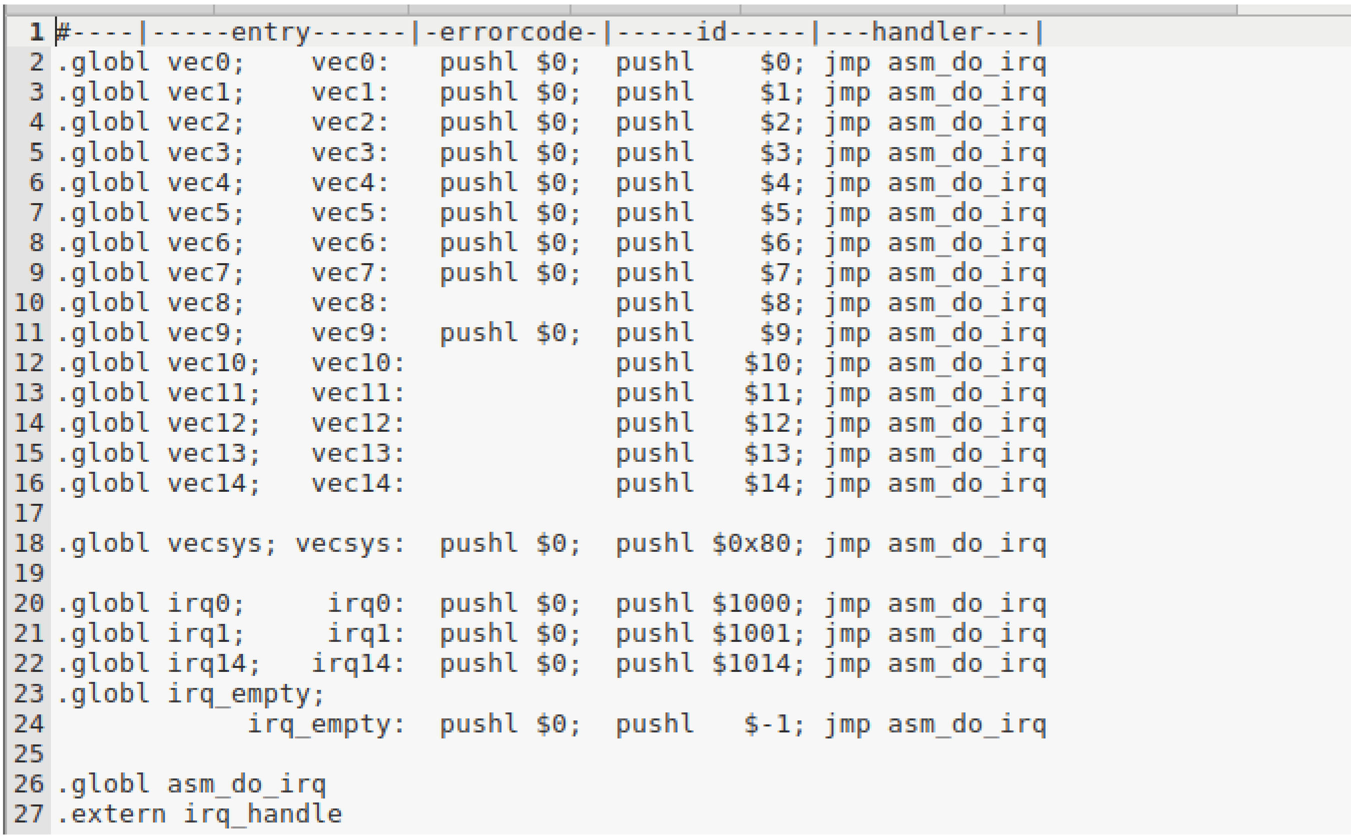
通过一个循环，每 0.01 秒系统都会发生一个信号给 nemu，每当信号来的时候，系统就会检测有没有事情发生，然后对于键盘事件，通过键盘的捕获 keyboard_up 和 keyboard_down，来监听键盘事件的产生。

5. 从键盘按下一个键到控制台输出对应的字符，系统的执行过程是什么？如果涉及与之前报告重复的内 容，简单引用之前的内容即可。



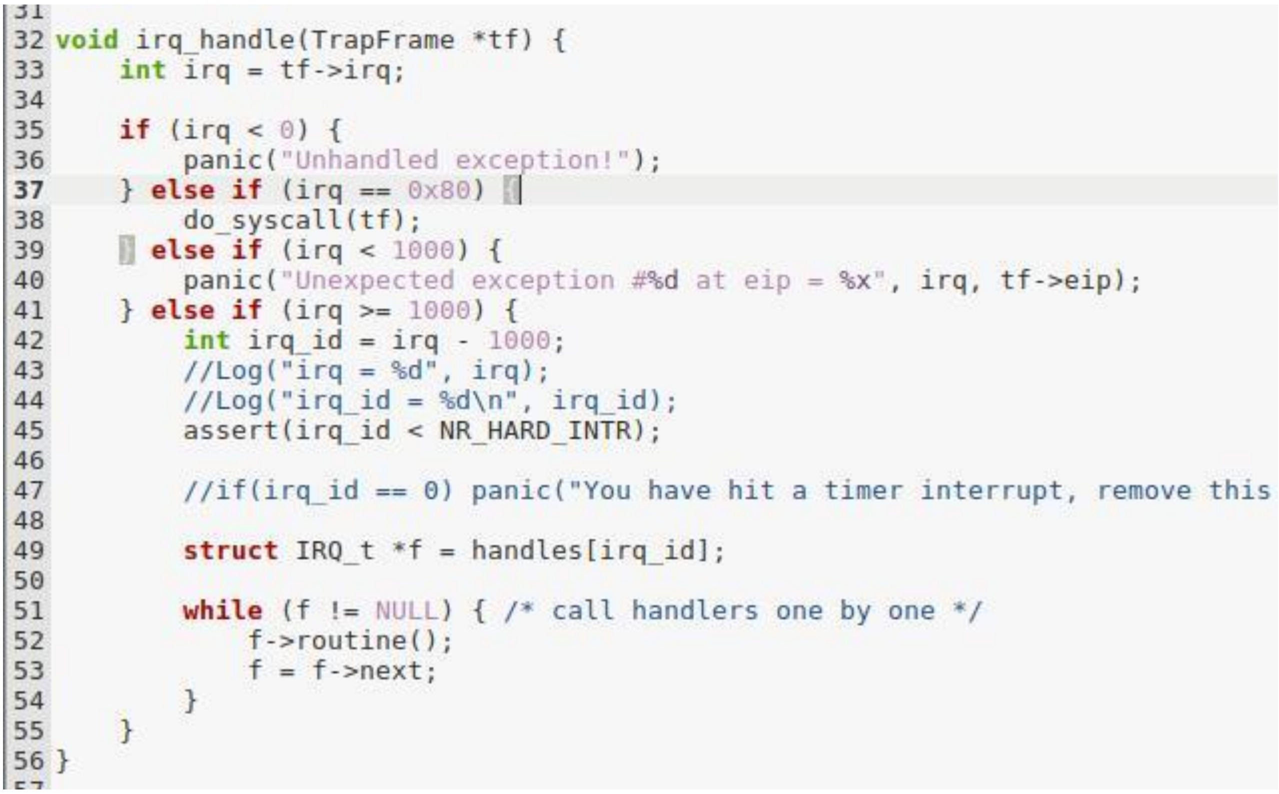
```
#include "device/i8259_pic.h"

static uint8_t scan_code_buf;

// called by the nemu_sdl_thread on detecting a key down event
void keyboard_down(uint32_t sym) {
    // put the scan code into the buffer
    scan_code_buf = sym2scancode[sym >> 8][sym & 0xff];
    // issue an iterrupt
    i8259_raise_intr(KEYBOARD_IRQ);
    // maybe the kernel will be interested and come to read on the data port
}

// called by the nemu_sdl_thread on detecting a key up event
void keyboard_up(uint32_t sym) {
    // put the scan code into the buffer
    scan_code_buf = sym2scancode[sym >> 8][sym & 0xff] | 0x80;
    // issue an iterrupt
    i8259_raise_intr(KEYBOARD_IRQ);
    // maybe the kernel will be interested and come to read on the data port
}

// called when the kernel issues an 'in' instruction on the keyboard's data port
make_pio_handler(handler_keyboard_data) {
    if(!is_write) {
        // only read allowed, and we do not consider race condition here
        write_io_port(port, len, scan_code_buf);
    }
}
```

利用前面的键盘事件监听功能去捕获键盘的扫描码，然后调用 i8259_raise_intr()来触发中断，然后去查表来调用相应的处理函数。



```
#----|-----entry------|-errorcode-|-----id-----|---handler---|
.globl vec0;    vec0:   pushl $0;  pushl    $0; jmp asm_do_irq
.globl vec1;    vec1:   pushl $0;  pushl    $1; jmp asm_do_irq
.globl vec2;    vec2:   pushl $0;  pushl    $2; jmp asm_do_irq
.globl vec3;    vec3:   pushl $0;  pushl    $3; jmp asm_do_irq
.globl vec4;    vec4:   pushl $0;  pushl    $4; jmp asm_do_irq
.globl vec5;    vec5:   pushl $0;  pushl    $5; jmp asm_do_irq
.globl vec6;    vec6:   pushl $0;  pushl    $6; jmp asm_do_irq
.globl vec7;    vec7:   pushl $0;  pushl    $7; jmp asm_do_irq
.globl vec8;    vec8:              pushl    $8; jmp asm_do_irq
.globl vec9;    vec9:   pushl $0;  pushl    $9; jmp asm_do_irq
.globl vec10;   vec10:             pushl   $10; jmp asm_do_irq
.globl vec11;   vec11:             pushl   $11; jmp asm_do_irq
.globl vec12;   vec12:             pushl   $12; jmp asm_do_irq
.globl vec13;   vec13:             pushl   $13; jmp asm_do_irq
.globl vec14;   vec14:             pushl   $14; jmp asm_do_irq

.globl vecsys; vecsys:  pushl $0;  pushl $0x80; jmp asm_do_irq

.globl irq0;     irq0:  pushl $0;  pushl $1000; jmp asm_do_irq
.globl irq1;     irq1:  pushl $0;  pushl $1001; jmp asm_do_irq
.globl irq14;   irq14:  pushl $0;  pushl $1014; jmp asm_do_irq
.globl irq_empty;
            irq_empty:  pushl $0;  pushl   $-1; jmp asm_do_irq

.globl asm_do_irq
.extern irq_handle
```

然后调用 irq_handle 进行处理



```c
void irq_handle(TrapFrame *tf) {
    int irq = tf->irq;

    if (irq < 0) {
        panic("Unhandled exception!");
    } else if (irq == 0x80) {
        do_syscall(tf);
    } else if (irq < 1000) {
        panic("Unexpected exception #%d at eip = %x", irq, tf->eip);
    } else if (irq >= 1000) {
        int irq_id = irq - 1000;
        //Log("irq = %d", irq);
        //Log("irq_id = %d\n", irq_id);
        assert(irq_id < NR_HARD_INTR);

        //if(irq_id == 0) panic("You have hit a timer interrupt, remove this

        struct IRQ_t *f = handles[irq_id];

        while (f != NULL) { /* call handlers one by one */
            f->routine();
            f = f->next;
        }
    }
}
```

这时就会调用显示的相关函数，将字符输出到屏幕上，最后再回到用户程序继续执行。