



南京大學

PA-4: 异常和中断 & 外设与 I/O & 游戏移植

课程名称: 计算机系统基础

姓名: 孙文博

学号: 201830210

邮箱: 201830210@smail.nju.edu.cn

实验时间: 2022.7.20 - 2022.7.31

一、实验目的

通过前三个阶段的 PA，我们已经基本构建了一个能够运算的机器的所有功能。目前为止，NEMU 只能够进行正常的控制流执行。在最后阶段，我们添加异常控制流的支持并使得 NEMU 能够实现和外设的 I/O。最终，我们希望在 NEMU 模拟器上能够运行类似仙剑奇侠传这样的小游戏。

二、实验过程

1. 异常和中断

a) 预备知识

从 80286 开始，Intel 统一把由 CPU 内部产生的意外事件，即，“内中断”称为异常；而把来自 CPU 外部的中断请求，即，“外中断”称为中断。而内部异常又分为三类：

1. 故障：与指令执行相关的意外事件，如“除数为 0”、“页故障”等；
2. 陷阱：往往用于系统调用；
3. 终止：指令执行过程中出现的严重错误。

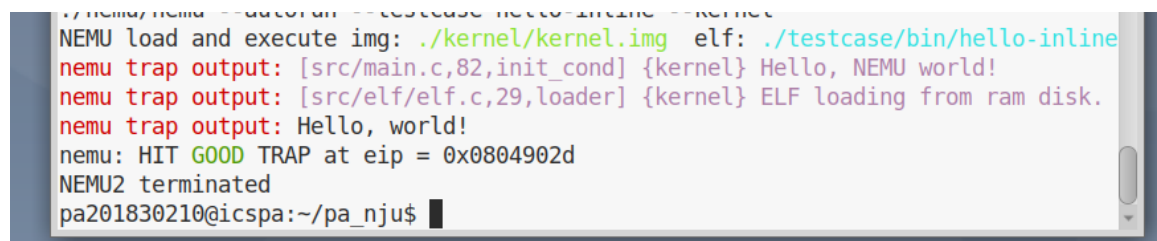
在本实验中，我们对于内部异常，只关注“陷阱”这一类。对于“故障”和“终止”这两类异常不做模拟，若遇到相应的情况，在 NEMU 中直接通过 `assert(0)` 强行停止模拟器运行。

异常和中断的响应和处理过程可分为两个阶段：第一阶段，CPU 对异常或中断进行响应，打断现有程序运行并调出处理程序；第二阶段，由操作系统提供的异常或中断处理程序处理完异常事件后返回用户程序继续执行。

b) 代码实现

§4-1.1 通过自陷实现系统调用

1. 在 include/config.h 中定义宏 IA32_INTR 并 make clean;
2. 在 nemu/include/cpu/reg.h 中定义 IDTR 结构体，并在 CPU_STATE 中添加 idtr;
3. 实现包括 lidt、cli、sti、int、pusha、popa、iret 等指令;
4. 在 nemu/src/cpu/intr.c 中实现 raise_intr()函数;
5. 执行 make test_pa-4-1 命令并看到屏幕输出:



```
./nemu/nemu --debug-on --testcase hello-inline --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/hello-inline
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu trap output: Hello, world!
nemu: HIT GOOD TRAP at eip = 0x0804902d
NEMU2 terminated
pa201830210@icspa:~/pa_nju$
```

§4-1.2 响应时钟中断

1. 在 include/config.h 中定义宏 HAS_DEVICE_TIMER 并 make clean;

2. 在 nemu/include/cpu/reg.h 的 CPU_STATE 中添加 uint8_t intr 成员, 模拟中断引脚;
3. 在 nemu/src/cpu/cpu.c 的 init_cpu() 中初始化 cpu.intr = 0;
4. 在 nemu/src/cpu/cpu.c 的 exec() 函数 while 循环体, 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;
5. 执行 make test_pa-4-1:

```

make-11: Leaving directory /home/pa201830210/pa_nju
./nemu/nemu --autorun --testcase hello-inline --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/hello-inline
nemu trap output: [src/irq/irq_handle.c,54,irq_handle] {kernel} system panic: You
have hit a timer interrupt, remove this panic after you've figured out how the c
ontrol flow gets here.
nemu: HIT BAD TRAP at eip = 0xc003129e
NEMU2 terminated
pa201830210@icspa:~/pa_nju$

```

6. 触发 Kernel 中的 panic, 找到该 panic 并移除。

```

49     else if (irq >= 1000)
50     {
51         int irq_id = irq - 1000;
52         assert(irq_id < NR_HARD_INTR);
53         if (irq_id == 0)
54             //panic("You have hit a timer interrupt, remove this panic after you've figured out how the control flow
gets here.");
55

```

移除后输出:

```

./nemu/nemu --autorun --testcase hello-inline --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/hello-inline
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu trap output: Hello, world!
nemu: HIT GOOD TRAP at eip = 0x0804902d
NEMU2 terminated
pa201830210@icspa:~/pa_nju$

```

2. 外设和 I/O

a) 预备知识

要完成与外设的 I/O, 核心要解决两个问题:

与谁进行 I/O?

I/O 的内容是什么?

对于第一个问题的回答涉及到 I/O 寻址的方式。一种 I/O 寻址方式是端口映射 I/O (Port-mapped I/O)。IA-32 共定义了 65536 个 8 位的 I/O 端口。在端口映射的 I/O 方式下, CPU 通过专门的 I/O 指令 `in (ins)` 和 `out (outs)` 来对某一个端口进行读和写。简言之, 通过对某一特定端口的读写, 就可以完成 CPU 和某特定外设之间的数据交换。至于交换的数据是控制命令、状态还是数据, 则不是 CPU 所关心的了。

另一种 I/O 寻址方式是内存映射 I/O (memory-mapped I/O)。这种寻址方式将一部分物理内存映射到 I/O 设备空间中, 使得 CPU 可以通过普通的访存指令来访问设备。这种物理内存的映射对 CPU 是透明的, CPU 觉得自己是在访问内存, 但实际上可能是访问了相应的 I/O 空间。这样以后, 访问设备的灵活性就大大提高了。一个例子是物理地址区间 `[0xa0000, 0xc0000)`, 这段物理地址区间被映射到 VGA 内部的显存, 读写这段物理地址区间就相当于对读写 VGA 显存的数据。例如:

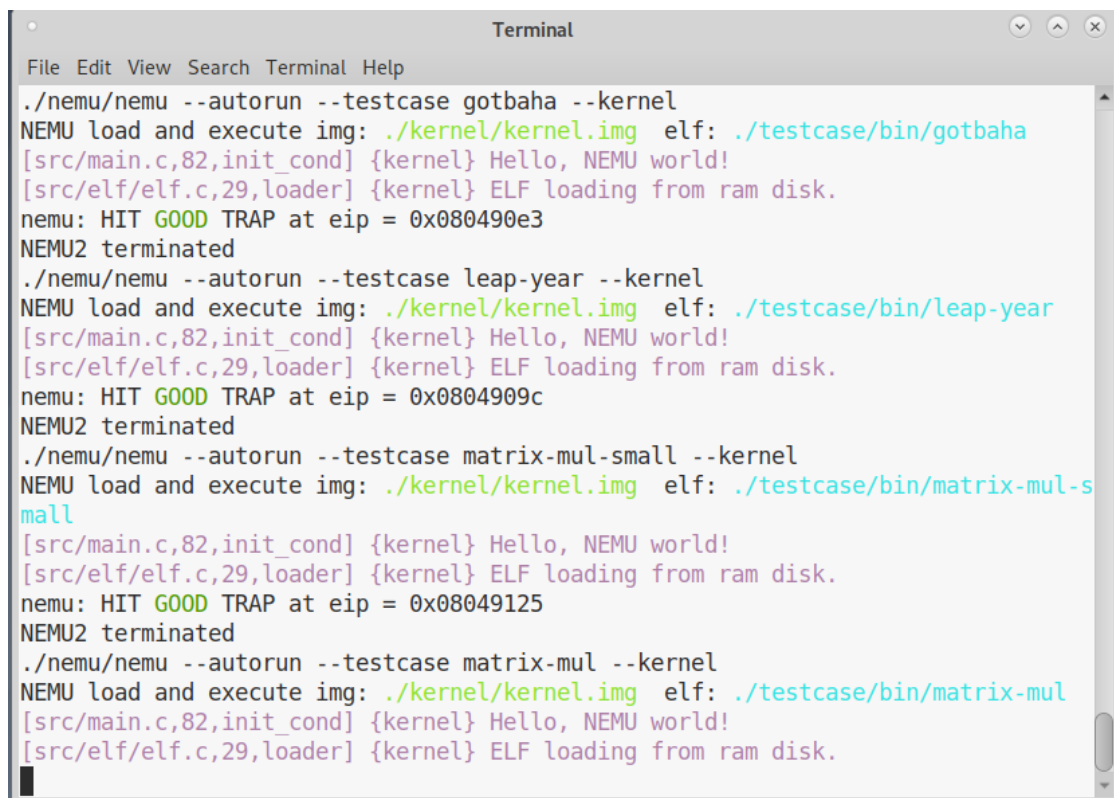
```
memset((void *)0xa0000, 0, SCR_SIZE);
```

会将显存中一个屏幕大小的数据清零, 即往整个屏幕写入黑色像素, 作用相当于清屏。

b) 代码实现

§4-2.3.1 完成串口的模拟

1. 在 include/config.h 中定义宏 HAS_DEVICE_SERIAL 并 make clean;
2. 实现 in 和 out 指令;
3. 实现 serial_printc()函数;
4. 运行 hello-inline 测试用例, 对比实现串口前后的输出内容的区别:



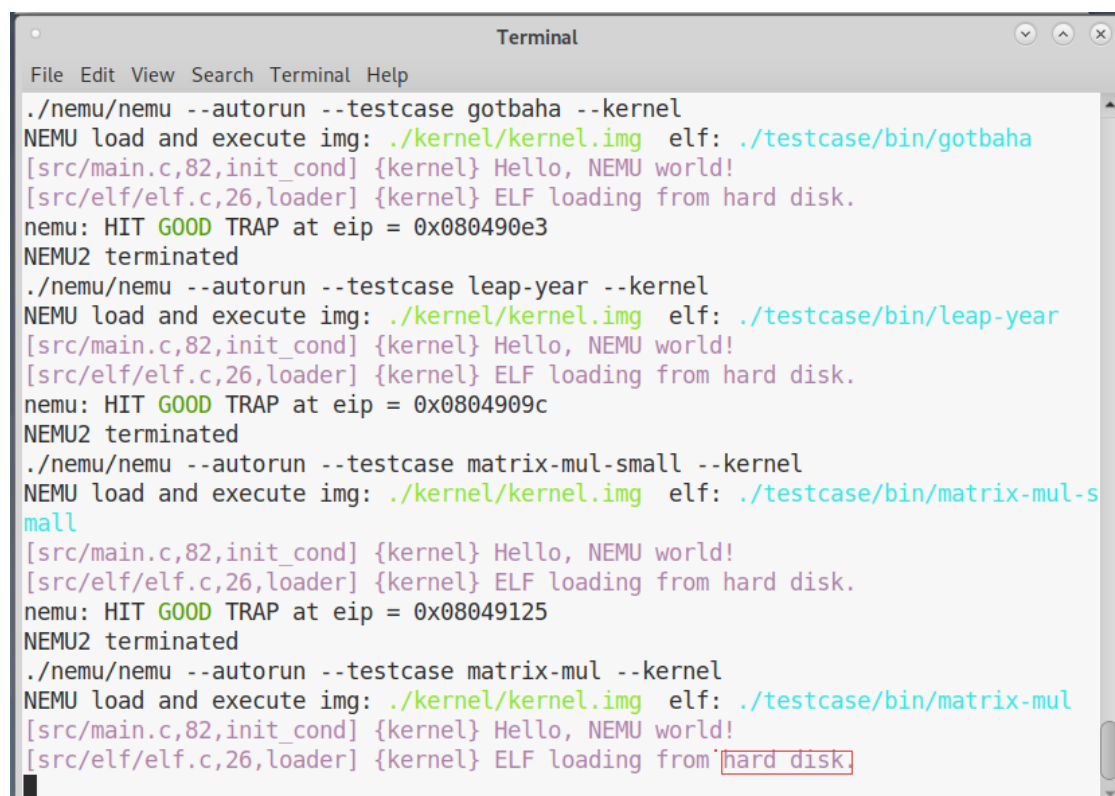
```
Terminal
File Edit View Search Terminal Help
./nemu/nemu --autorun --testcase gotbaha --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/gotbaha
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x080490e3
NEMU2 terminated
./nemu/nemu --autorun --testcase leap-year --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/leap-year
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x0804909c
NEMU2 terminated
./nemu/nemu --autorun --testcase matrix-mul-small --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/matrix-mul-small
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x08049125
NEMU2 terminated
./nemu/nemu --autorun --testcase matrix-mul --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/matrix-mul
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
```

5.

此时不再通过 NEMU trap 输出。

§4-2.3.2 通过硬盘加载程序

1. 在 include/config.h 中定义宏 HAS_DEVICE_ID 并 make clean;
2. 修改 Kernel 中的 loader(), 使其通过 ide_read()和 ide_write()接口实现从模拟硬盘加载用户程序;
3. 通过 make test_pa-4-2 执行测试用例, 验证加载过程是否正确:



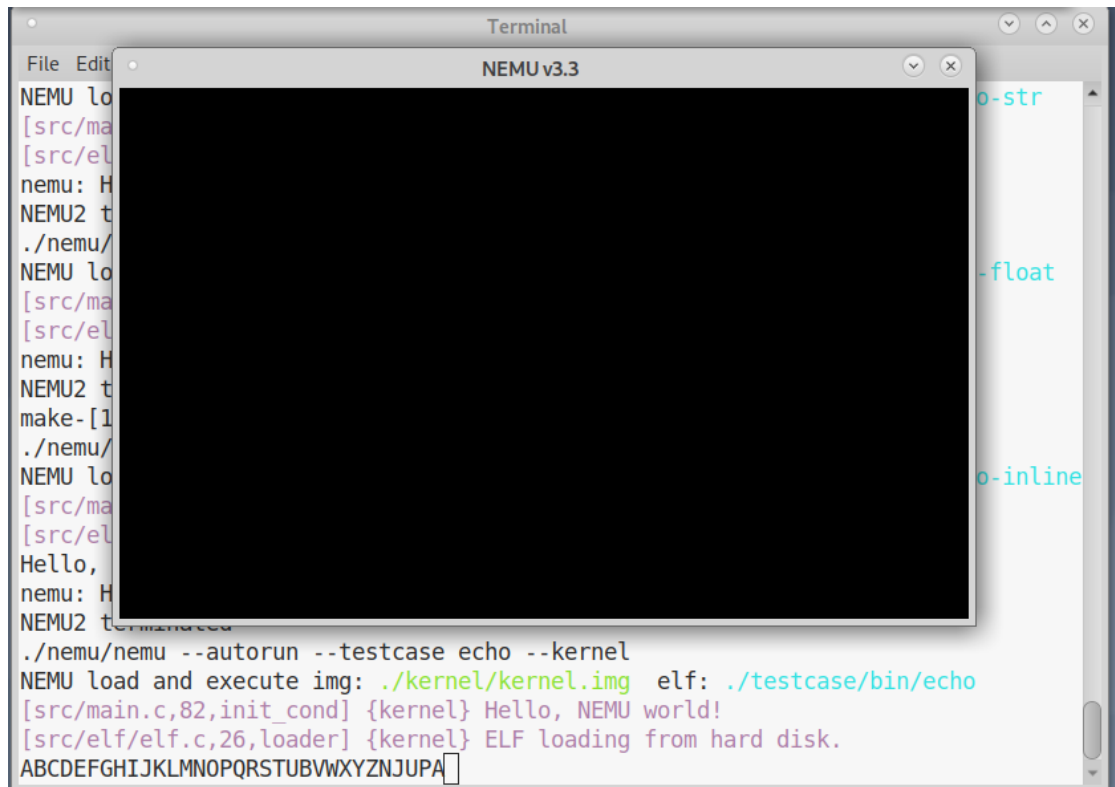
```
Terminal
File Edit View Search Terminal Help
./nemu/nemu --autorun --testcase gotbaha --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/gotbaha
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,26,loader] {kernel} ELF loading from hard disk.
nemu: HIT GOOD TRAP at eip = 0x080490e3
NEMU2 terminated
./nemu/nemu --autorun --testcase leap-year --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/leap-year
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,26,loader] {kernel} ELF loading from hard disk.
nemu: HIT GOOD TRAP at eip = 0x0804909c
NEMU2 terminated
./nemu/nemu --autorun --testcase matrix-mul-small --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/matrix-mul-small
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,26,loader] {kernel} ELF loading from hard disk.
nemu: HIT GOOD TRAP at eip = 0x08049125
NEMU2 terminated
./nemu/nemu --autorun --testcase matrix-mul --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/matrix-mul
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,26,loader] {kernel} ELF loading from hard disk.
```

此时 ELF 从模拟硬盘加载。

§4-2.3.3 完成键盘的模拟

1. 在 include/config.h 中定义宏 HAS_DEVICE_KEYBOARD 并 make clean;

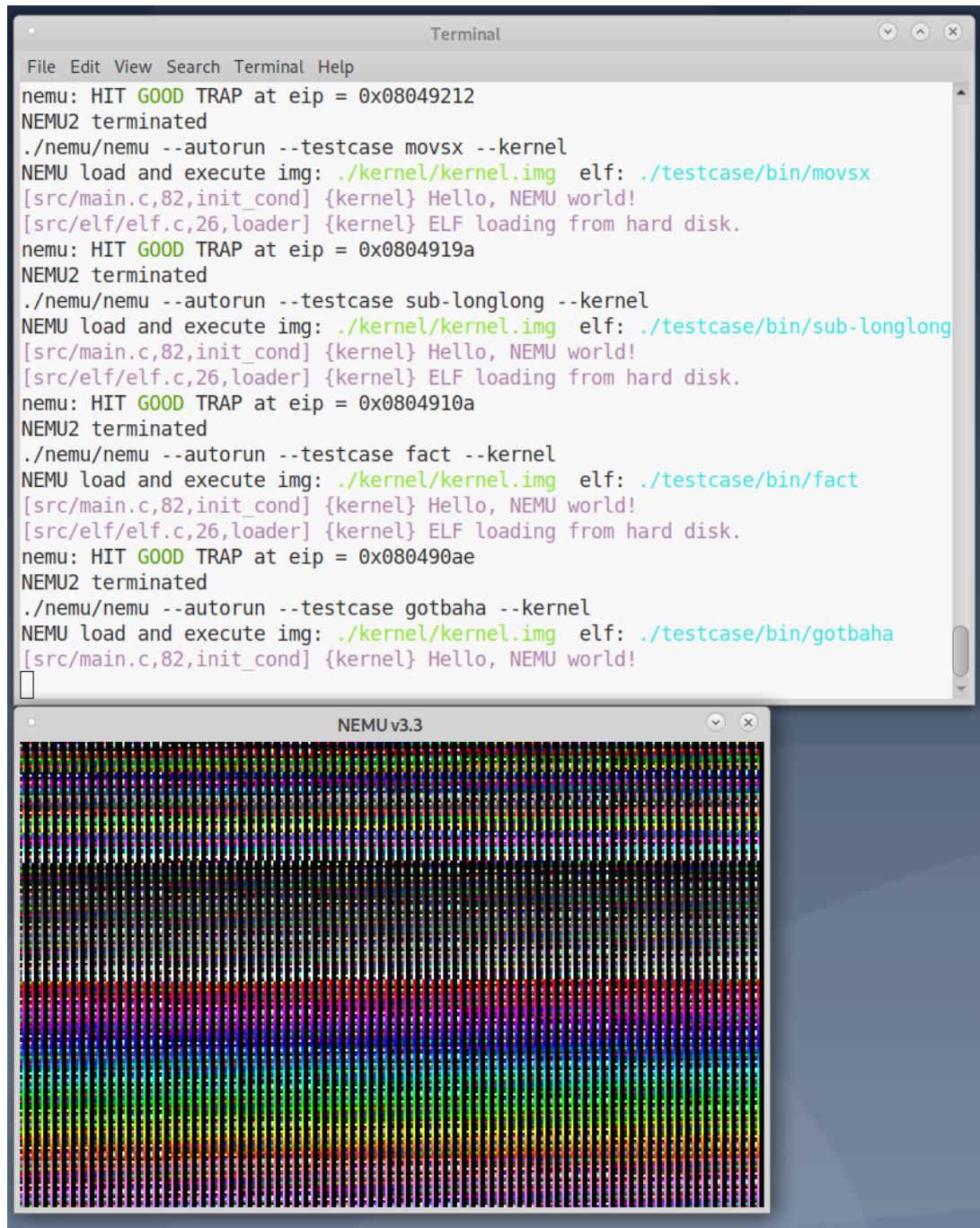
2. 通过 `make test_pa-4-2` 运行 echo 测试用例; (可以通过关闭窗口或在控制台 Ctrl-c 的方式退出 echo) :



此时 NEMU 实现的功能是输出键盘上键入字母的大写模式。

§4-2.3.4 实现 VGA 的 MMIO

1. 在 `include/config.h` 中定义宏 `HAS_DEVICE_VGA`;
2. 在 `nemu/src/memory/memory.c` 中添加 `mm_io` 判断和对应的读写操作;
3. 在 `kernel/src/memory/vmem.c` 中完成显存的恒等映射;
4. 通过 `make test_pa-4-2` 执行测试用例, 观察输出测试颜色信息, 并通过 `video_mapping_read_test()`:

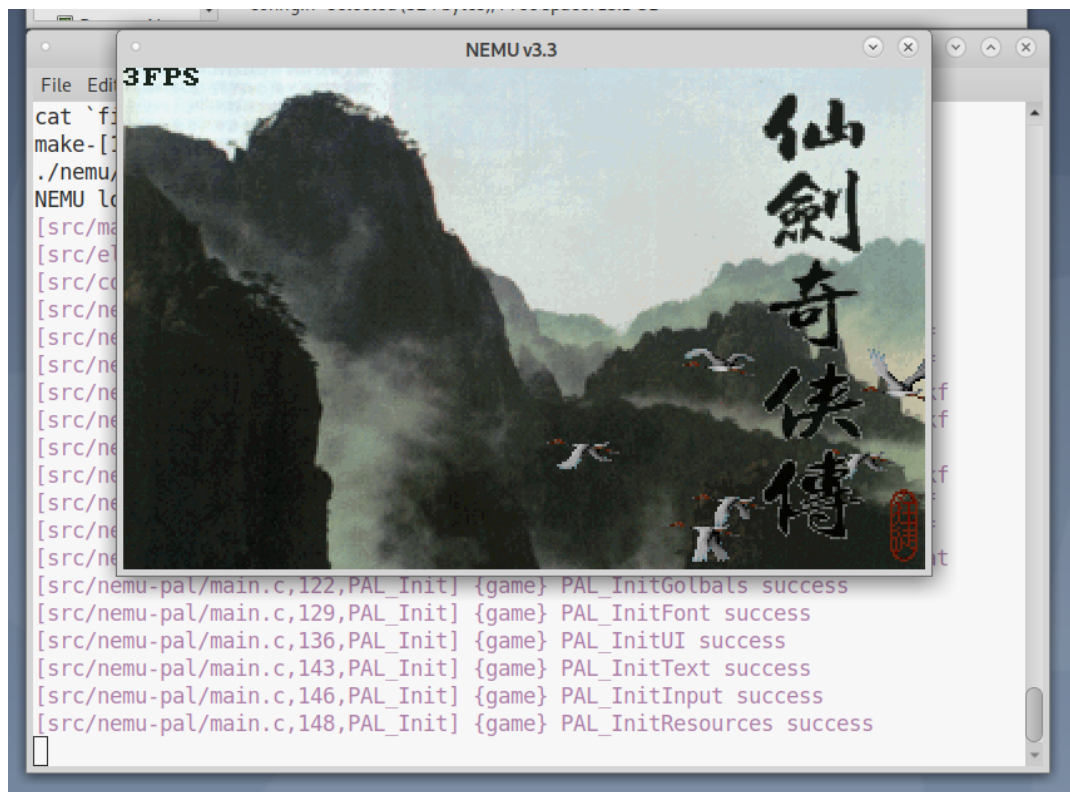


The image shows two overlapping windows. The top window is a terminal titled 'Terminal' with a menu bar (File, Edit, View, Search, Terminal, Help). It displays the output of running NEMU tests. The output shows four successful test runs for 'movsx', 'sub-longlong', 'fact', and 'gotbaha', each followed by 'NEMU2 terminated'. The bottom window is titled 'NEMU v3.3' and displays a corrupted, multi-colored screen, likely representing a game that failed to render correctly.

```
File Edit View Search Terminal Help
nemu: HIT GOOD TRAP at eip = 0x08049212
NEMU2 terminated
./nemu/nemu --autorun --testcase movsx --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/movsx
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,26,loader] {kernel} ELF loading from hard disk.
nemu: HIT GOOD TRAP at eip = 0x0804919a
NEMU2 terminated
./nemu/nemu --autorun --testcase sub-longlong --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/sub-longlong
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,26,loader] {kernel} ELF loading from hard disk.
nemu: HIT GOOD TRAP at eip = 0x0804910a
NEMU2 terminated
./nemu/nemu --autorun --testcase fact --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/fact
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
[src/elf/elf.c,26,loader] {kernel} ELF loading from hard disk.
nemu: HIT GOOD TRAP at eip = 0x080490ae
NEMU2 terminated
./nemu/nemu --autorun --testcase gotbaha --kernel
NEMU load and execute img: ./kernel/kernel.img elf: ./testcase/bin/gotbaha
[src/main.c,82,init_cond] {kernel} Hello, NEMU world!
```

3. 游戏移植

根据手册完成了两款游戏的移植，其中包括对文件系统的模拟等等步骤，遗憾的是打字小游戏可以正常运行，但是仙剑奇侠传还是卡在了菜单界面（应该是 NEMU 性能不够），以后有时间



三、思考题

§4-1.3.1 通过自陷实现系统调用

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；

答： 执行 `int $0x80` 时，调用了 `int` 指令，通过解析操作码，获取中断号 `0x80`， 随后将其作为参数，调用 `raise_sw_intr()`函数，该函数更新 `eip` 地址后，便调用 `raise_intr()`函数。

在 `raise_intr` 函数中的 `intr_no` 依然是 `0x80`。随后, 依次将 `eflags`, `CS` 和 `eip` 的值压栈, 并从 `IDTR` 总读出 `IDT` 的首地址, 根据中断号 `0x80` 在 `IDT` 中索引得到一个门描述符, 把门描述符的段选择符装载入 `CS` 寄存器, 接着调用 `load_sreg` 函数加载 `cs` 的隐藏部分。根据段选择符中 `type` 的信息判断 是中断还是陷阱。如果是中断便把 `IF` 清零。最后把 `offset` 赋给 `eip`, `raise_intr` 的使命也就终结了。

随后返回 `int` 指令, 由于 `return 0`, 此时的 `eip` 便是中断处理程序的入口地址。执行到这一步后, 便是操作系统 (kernel) 的工作了通过入口地址的信息, 跳转到 `kernel/src/irq/do_irq.S` 的入口函数 `vecsyzs()`, 执行 `pushl 0x80` 后, 压入错误码和异常号, 跳转到 `asm_do_irq` 中, 执行三个阶段:

准备阶段: 执行 `pushal` 和 `pushl %esp`, 在内核栈中保存各寄存器内容 (现场信息)。代码将会把用户进程的通用寄存器保存到堆栈上, 这些寄存器的内容 连同之前保存的错误码, 以及 `eflags`, `CS`, `eip` 形成了 `trap frame`

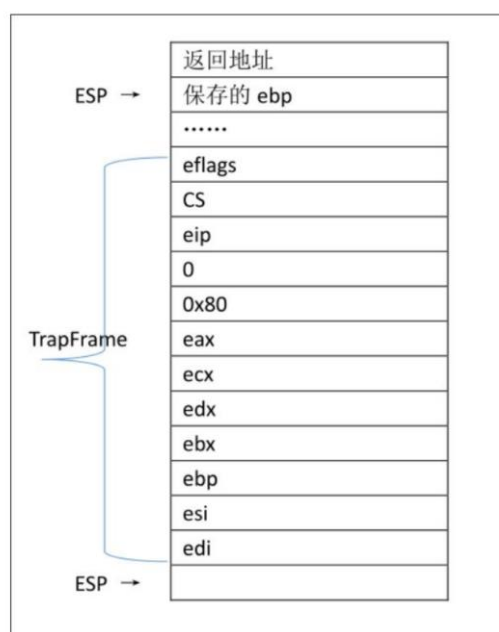
处理阶段: 执行 `call irq_handle`, 调用函数 `irq_handle`, 此时传入的是 `TrapFrame` 的变量 `*tf`, 根据 `tf` 读出 `irq` 确定异常事件的类型, 由于是 `0x80`, `kernel` 调用 `do_syscall()` 函数。在 `do_syscall()` 函数中, 根据 `hello-inline` 中传入的参数 `tf->eax`, 为 `4`, 因此调用 `sys_write` 函数, 该函数根据 `tf` 指针把 `TrapFrame` 的 `ebx`, `ecx`, `edx` 传入 `fs_write()` 函数调用, 从而在屏幕输出 `hello world`。

恢复阶段: 返回到 `do_irq.S` 中, 执行 `popa` 和 `iret`, 是恢复用户进程的现场, `kernel` 将根据之前保存的 `trap frame` 中的内容, 恢复用户进程的通用寄存器, 最后通过 `iret` 指令恢复用户进程的 `eip`, `CS`, `eflags`。系统调用结束此后, `cpu.eip` 回到 `int` 指令的后一条指令, 便继续执行 `hello-inline` 的代码。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。

答: (1) `push %esp` 的作用是把执行完 `pusha` 后的 `esp` 压栈, 而这个 `esp` 指向的是 `TrapFrame` 的首地址, 因此这个步骤是在把 `TrapFrame` 的指针作为参数传给 `irq_handl`。

(2) 在 `call irq_handle` 之前, 系统栈的内容如下:



§4-1.3.2 响应时钟中断

1. 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。

答: 首先 `raise_intr` 做的是把 EIP 指向 kernel 代码, 真正处理时钟中断是在 kernel 代码, `raise` 函数的作用就是把操作系统喊过来。

有一个函数指针指向的函数会处理, `irq_handle.c` 里面创建了一个关于系统调用的链表数组, 链表结构, 但是每一个元素对应一个系统调用号的数组。

相同之处: 整个过程就是 `push` 现场之后, 在 `irq_handle.c` 里面看是哪个系统调用; 都是通过 `int $80` 这样的形式陷入 kernel 层次的代码(只是模拟层面是用 C 语言的 `if-else`, 本质上是一样的), 由 kernel 来处理中断。

不同之处: 我们观察 `HIT_BAD_TRAP` 的地方 (如下), 可以看到我们模拟的 C 代码层面的陷入内核代码, 即调用 `do_syscall` 函数, `push` 现场之后。

```
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;
        assert(irq_id < NR_HARD_INTR);
        if(irq_id == 0) {
            // panic("You have hit a timer interrupt, remove this panic after you've figured out how the
        }

        struct IRQ_t *f = handles[irq_id];

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

如果是时钟中断, 我们发现这里有一个 panic:

```
#define panic(format, ...) \
do { \
    cli(); \
    Log("\33[1;31msystem panic: " format, ## __VA_ARGS__); \
    HIT_BAD_TRAP; \
} while(0)
```

就是强制 HIT_BAD_TRAP 了, 这是不同的地方。

§4-2.3. 外设与 I/O

针对 echo 测试用例，在实验报告中，结合代码详细描述：

1. 注册监听键盘事件是怎么完成的？

答：开启 HAS_DEVICE_KEYBOARD 后，在 testcase/srt/echo.c 中，main 函数通过调用 add_irq_handler，将 IRQ_t 类型的指针存入 handles 数组，从而完成注册监听键盘事件。

2. 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。

答：对键盘展开模拟时，键盘事件首先在 nemu/src/device/sdl.c 中由 NEMU_SDL_Thread() 线程捕获。当检测到相应事件后，将对应键的扫描码作为参数传送给 keyboard.c 中的模拟键盘函数。模拟键盘缓存扫描码，并通过中断请求的方式通知 CPU 有按键或抬起的事件，键盘的中断请求号为 1。

CPU 收到中断请求后调用 Kernel 的中断响应程序。在响应程序中，Kernel 会查找是否有应用程序注册了对键盘事件的响应，若有，则通过调用注册的响应函数的方式来通知应用程序。此时在应用程序的键盘响应函数中，可以通过 in 指令从键盘的数据端口读取扫描码完成数据交换。键盘数据端口约定为 0x60，键盘扫描码的编码方式参照这个约定。

四、实验总结

长达一个学期的 PA 实验最终还是要落下帷幕了，当然我们在这半年多的时间里见识到的也不过是 NEMU 的一部分，这里不得不再次感谢创始人 yzh 学长及他的团队，感谢参与实验的老师 and 助教们，单独夸一夸 wl 男神真的很能体谅我们的时间安排，不到最后一刻永远可以慢慢的做自己的 PA。相识即缘分，这一路走来确实不容易，未来也不会忘记。那么就祝老师们和学长的事业可以蒸蒸日上，也希望未来自己可以从事一份喜欢的工作吧！❤